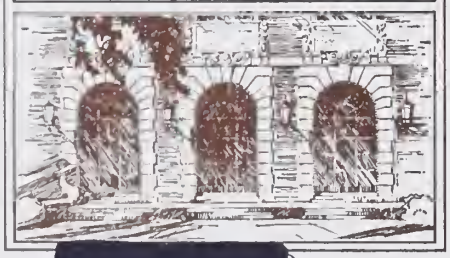
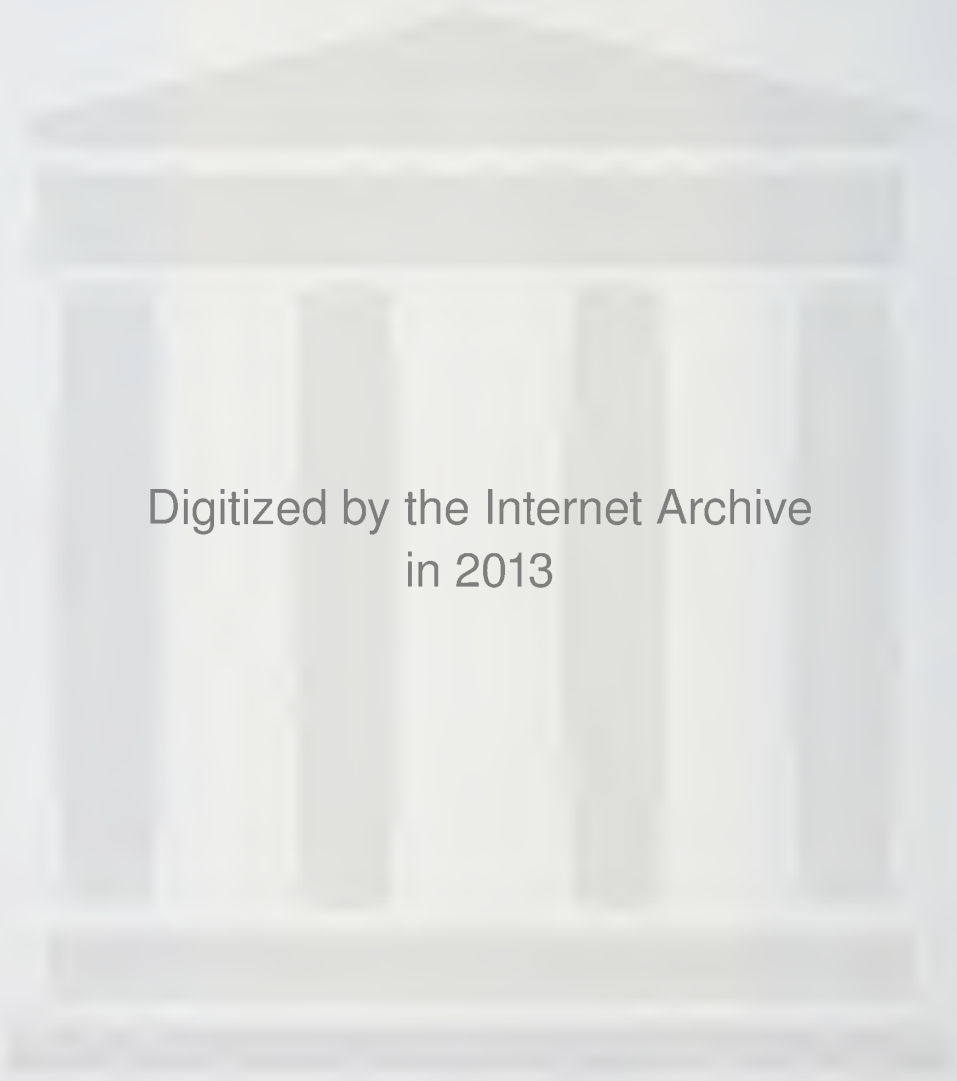




UIUCDCS-R-76-797

A STUDY IN THE DESIGN OF AN ARITHMETIC ELEMENT FOR SERIAL PROCESSING IN A LINEAR ITERATIVE STRUCTURE

by

Lakshmi Narayana Goyal

May, 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-76-797

A STUDY IN THE DESIGN OF AN ARITHMETIC ELEMENT FOR SERIAL PROCESSING IN A LINEAR ITERATIVE STRUCTURE

by

Lakshmi Narayana Goyal

May, 1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was supported in part by the Department of Computer Science and in part by the National Science Foundation under grant NSF DCR 73-07998, and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering, 1976.

ACKNOWLEDGEMENTS

I wish to express my deep gratitude to my thesis advisor, Professor James E. Robertson, for his invaluable guidance, insight, constant encouragement and personal friendship during the preparation of this thesis. I would also like to thank Professors D. J. Kuck and S. R. Ray for their advice, friendship and time for many useful discussions.

The support of the Department of Computer Science of the University of Illinois, the Atomic Energy Commission of the United States and the National Science Foundation during my studies at the University of Illinois is sincerely appreciated. Many thanks are due to Mr. Stan Zundo of the drafting department for the excellent illustrations, his personal friendship and constant cooperation and to Mr. Dennis Reed for fast and excellent printing services. The excellent cooperation of Mr. Mark Goebel of the drafting department is very much appreciated.

Finally, I wish to thank my wife, Madhu, for her patience, understanding, love and constant encouragement without which this thesis would not have been possible.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

APR	Adjacent Operand digit Register
AR	"Assimilation Recode' Microinstruction
BU	Borovec Unit - A 2-input redundant binary adder
DMM	Data Main Memory
DPG	Digit Product Generator
DPL	Digit Processing Logic
DSE	Digit Sum Encoder
ECU	Exponent Control Unit
EPL	Exponent Processing Logic
FMA	'Form Multiple and Add' Microinstruction
IBR	Register File Input Bus Buffer Register
INR_i	ith Internal Register (File Register)
GACU	Global Arithmetic Control Unit
GIR	G-information Input Buffer Register
LDR	LOMM Data Register
LOEM	Local Operand Exponent Buffer Memory
LOMCO	Local Operand Memory Controller
LOMM	Local Operand Mantissa Buffer Memory
LPM	'Load PE from PEM' Microinstruction
LVE_i	ith Logic Vector Encoding (i=1,2,3)
MATD	Multi-input Adder's 'Transfer' Decoder
MATE	Multi-input Adder's 'Transfer' Encoder
MCU	Mantissa Control Unit
MIAD	Multiinput Adder Network
MIR	Microinstruction Register
MIRBA	Multiinput Redundant Binary Adder
MPL	Mantissa Processing Logic
MS	'Multi Sum' Microinstruction
NR	'Normalization Recode' Microinstruction
PE	Processing Element
PEM	Processing Element Memory
RBA-2	Two input Redundant Binary Adder

List of Abbreviations (continued)

RBA-3	Three input Redundant Binary Adder
RB_r	Redundant Binary Encoded Format for Radix-r digit
RIP_i	Register Input Port for ith Processing Element
ROP_i	Register Output Port for ith Processing Element
RS	'Right Shift' Microinstruction
SM_b	Sign Magnitude logic encoding format for a redundant binary digit
SM_r	Sign Magnitude Encoding format for a radix-r signed digit
SPM	'Store into PEM from PE' Microinstruction
sRIB	Selector for Register File Input Bus
sROB	Selector for Register File Output Bus
SS	'Simple Sum' Microinstruction
sTOP	Selector for 'Transfer Output Port'
sADR	Selector for data input to Adder Network MIAD
sDSE	Selector for input to Digit Sum Encoder, DSE
TD	'Transfer Direct' Microinstruction
TI	'Transfer with Inverted Sign' Microinstruction
TIP_i	Adder Transfer Input Port for ith PE
TOP_i	Adder Transfer Output Port for ith PE

1. INTRODUCTION

1.1 Introduction to LSI Design Constraints

The advent of large scale integration (LSI) technology for the manufacture of logic circuits has posed a new challenge to the computer system and logic designers. The challenge is to find out ways that would make efficient use of the full potential of LSI--reliability, lower cost and improved speed--in the design of digital computers.

The LSI technology has peculiar constraints which have important implications for its effective use in future systems. The constraints and implications can be broadly classified into two categories: external or system level considerations and internal or logic circuit level constraints. With LSI, hundreds of logic functions can be fabricated on subminiature substrates. Since the initial development cost is very high, it is important that a small number of standard elements be developed and the initial cost of development, thus, gets amortized. However, designing universal elements of the complexity offered by LSI is very difficult. A potential benefit of LSI that has been continually cited is an increase in reliability over current systems. Since system reliability is inversely proportional to the number of module interconnections, it is important that LSI devices should have a high gate-to-pin ratio. But the idea of universality of LSI devices and high gate-to-pin ratio are conflicting in that the latter tends to give the device a unique personality and cannot be used in a system repetitively.

At the internal level, designing logic for integration on the chip requires a reorientation of the relative values placed on the resources used to realize the design. One of the severest constraints in the design of an LSI device is the restriction on interconnections on the chip itself. This is due to the limitation both of available wiring area, the number of planes to which all of the wiring must be confined, and a host of other topological considerations which combine to determine the locations of candidate points for interconnections. Required wiring can be reduced by forcing the logic design of the chip into a cellular or regular structure. Regular structure has very important implications.

a) It facilitates every step of LSI manufacturing process by making it possible to perform relatively simple tasks repetitively. Mask making can be facilitated by the repetitive structure.

b) It is possible to design and optimize a simple cell to achieve most function per dollar, but a large chip of random gates is impossible to optimize because of variables involved.

c) Testing of LSI devices is a major cost factor. The generation of test algorithms for simple cells and regular and repetitive structure is easier.

d) In addition, as the yield increases due to technology improvements, larger devices can be made out of the same simple cells.

Another limitation of LSI which must be considered in logic design is that of external connections. More pins require more external gates to drive the capacitance of the external pins. It causes an increase in temperature due to increased number of gates and higher current required due to a large number of external gates. This increases the failure rate of the device.

Semiconductor memories meet most of the requirements of LSI technology and the present use of LSI in computer systems is in the form of these memories for system enhancement applications [1],[2],[3]. These applications include the use of RAMs as scratch-pad memories, of cache memories to reduce main memory requests and, thus, increase the computational throughput. Use of ROMs for microprogramming, table look-up operations and hardwired subroutines also increase performance at a relatively little cost. Content addressable memories, queues and stacks can greatly simplify building and maintaining tables and greatly reduce system overhead and software costs.

Use of LSI in the design of central processor itself involves proper logic partitioning. Logic partitioning involves organizing the internal logic structure such that large functional arrays on a chip can be repetitively used. Two partitioning methods are the bit-slicing and functional partitioning. Bit-slicing [4],[5] tends to be system dependent and not universal and, thus, is suitable for custom LSI only. In functional partitioning, the machine is structured towards modules wherein each module consists of a completely self-contained processor having local storage, some processing logic and the control necessary for the module to execute its function. Each module acts as a small

insular unit of logic. The module control sees only its own state and the requirements for communication outside the module are correspondingly reduced. An excellent example of functional partitioning is RCA's LIMAC and Macromodular computers [6],[7].

In this thesis, we report on a study of logical organization and design of an Arithmetic Unit which is capable of performing four basic operations of addition, subtraction, multiplication and division.

The organization and design of the Arithmetic Unit are influenced by LSI technology constraints of modularity, least number of different module types, structural regularity of the module, limited pin count and limited fan-out capability.

In the rest of this chapter, we first very briefly review the various proposals suggested for the Arithmetic Unit and its LSI implementation. This is followed by a brief introduction of the model chosen for study in this research and the scope and an overview of the thesis.

1.2 Arithmetic Unit Structure and LSI

The proposals suggested for the architecture of LSI implementable Arithmetic Units can be broadly classified into three categories; namely: a) partitioning of the conventional ALU which uses standard binary number representation, b) two dimensional iterative (cellular) structures and table look-up methods, and c) use of number representations different than conventional binary. It must be noted that these three categories are not exclusive of each other but are rather interrelated. This classification is used here simply for ease of exposition.

1.2.1 Partitioning of conventional ALU - A low performance and parallel basic ALU essentially consists of registers--the accumulator, the M-Q register, and the registers for parallel shifting--a full adder, circuitry for complementing and shifting and some control logic to coordinate their activity for arithmetic and logic control. It is necessary for efficient operation to allow flexible and rapid transfer of information from any one register to another. In binary arithmetic, except for the end bits, it is possible to partition all the circuitry associated with one bit along with some local decoder bits for gating functions into one LSI cell [8]. Thus, all the data transfer and manipulation operations circuitry can be assembled into identical cells and provide a fairly good gate/pin ratio. A classic example of this approach is the Texas Instrument LSI airborne computer, the model 2502. However, this bit-slicing approach breaks down for high performance, conventional binary number system when circuitry for a fast carry generation and propagation is added to the ALU. Raytheon [9] has combined four bit slices into one LSI module so that the look-ahead logic could be used on the four bits of this module. But still this does not provide the flexibility for unlimited carry look-ahead with only one type of module.

For achieving high performance, with conventional number systems, the various slices of the ALU work in synchro-parallelism [10] and controlled by signals broadcast from the central control logic. Since the control functions are more difficult to modularize than functions related to data operations, micromemory control technique is used for mapping the irregular and diverse algorithms for arithmetic control into a

regular structure of memory. However, for large word length, the broadcasting of control signals is not compatible with LSI constraint of low fan-out and neighborhood connections only.

To overcome this problem of control irregularity and broadcasting, many combinational two-dimensional iterative structures (cellular arrays) have been proposed for multiplication, division and other arithmetic functions like square root, etc.

1.2.2 Two dimensional iterative structures - Two dimensional iterative structures are memory-like structures and admirably satisfy the LSI constraints. From the arithmetic unit point of view, they can be further classified into two sub-categories of cellular arrays and table look-up methods.

1.2.2.1 Cellular arrays - A cellular array is a two-dimensional iterative configuration of identical cells, each of which contains both logic and storage and is connected mainly to its immediate neighbors. Such an array, therefore, has the form of a memory array that is enhanced with logic at each digit position. A cellular array is a spatial analog of the temporal sequence of steps of the control algorithm; i.e., the cellular array performs the same sequence of computations iteratively in space rather than in time. The cellular arrays can be either purely dedicated exclusively [11] to some arithmetic function or can be programmable [12] so that they can be used by many functions. Since multiplication processes are characterized by the basic algorithm of add/no add followed by shift, they differ mainly in the interconnection of the various cells in the array for speeding up the effective addition

time of the various partial products. Some use tree adder structure [13] while others use carry save adders [14] in the basic cells to avoid the carry propagation problem at every stage--the carry propagation occurring only at the last stage. Most arrays assume that the operands are either positive or in the sign magnitude representation with the sign of the product being determined separately. A negative multiplier in 2's complement representation needs a correction to the product obtained by the simple "add/no add and shift" algorithm and makes the interconnection of the cells in the array somewhat irregular. A cellular array for multiplication has been suggested [15] which makes use of multiplier recoding and a conditional adder/subtracter cell so that either addition or subtraction of the shifted multiplicand to the partial product can take place. This does not require final correction to the product. However, the recoding array is structurally different from the multiplication array and thus needs two types of functional arrays to generate the final product. More recently, Baugh and Wooley [16] have proposed a cellular multiplier where the correction is not necessary. Similarly, the cellular array for the division operation uses the basic binary restoring or non-restoring algorithm to produce the quotient. The interconnection structure for the end cells used for comparing the signs of the divisor and the partial remainder is again different from the rest of the cell interconnection [17] . For fixed point operations, a special step may be necessary at the end to generate the remainder of the same sign as the dividend. In addition to the dedicated arrays for each arithmetic operation, a programmable array suitable for both multiplication and division has also been recently

proposed. Here the most significant cell of each row is more complicated and acts both as a multiplier recoder cell and a comparator cell for comparing the signs of the divisor and partial remainders, depending upon the arithmetic operation [18]. For operands of large word length, the cellular array contains more cells than can reliably be implemented on a single silicon slice, and hence is subdivided into subarrays which are externally connected. Such an array made up of subarrays will take more time to generate the final result compared to a fully iterative monolithic array on a single silicon slice. Cellular arrays can be either synchronous or asynchronous in operation. An asynchronous cellular multiplier for vector or pipelined mode operations has been proposed by Bjorner [19].

For arrays using conventional number systems, the problem of carry propagation along a row of cells still plagues the cellular arrays.

Thompson [20] and Chen [10] have suggested using the cellular array in a diagonally-timed fashion such that digit level pipeline takes place in two dimensions, giving a higher computational throughput.

Although the cellular arrays do satisfy the structural needs of logic circuits, for LSI technology, a few shortcomings can be summarized as follows:

i) Due to the use of conventional ripple carry adder/subtracter in the basic cells, the total number of cells needed is always equal to that necessary for a double length product, although in practice most often the single length product only is desired. This is due to the fact that the carry

ripples from the least significant end to the most significant bit position and if the cells that contribute to the less significant part of the double length product (for fractional mantissas) do not exist, then the most significant part of the product will be in error and this error becomes very acute for operands of large word length. These otherwise unnecessary cells raise the cost of the array.

ii) In the case of cellular arrays, as many rows of adder/subtracter basic cells are needed as there are multiplier bits or quotient bits. But effectively, the rows corresponding to "zero" multiplier or quotient bits serve no useful purpose (except possibly shifting). These unnecessary cells not only add to the cost by using large amounts of silicon area, but they also increase the probability of faults on the chip and make the testing of the chip more expensive.

iii) For large word length where subarrays have to be externally connected, the addition of other subarrays for expansion of word length cannot be done without extensive changes in the external wiring. Thus expandability of two-dimensional structures is poor.

1.2.2.2 Table look-up methods - Structural regularity of memories makes them very suitable for implementation in large-scale integration. All the logic and arithmetic operations in the machine can be performed

by extensive table look-up operations. Table look-up operations can either be done in parallel or in a serial fashion. Parallel operations, however, require too large a table for any reasonable word length and are out of the question. However, tables for bit parallel and byte serial operations can be reasonably implemented for arithmetic operations like addition and multiplication because the number of words required is related to 2^{2n} where n is the operand width in bits. A functional memory based on an associative array composed of writeable storage cells capable of holding three states--0, 1, and don't care--has been proposed by Gardner [21]. Here the logic is performed by associative table look-up and uses the "don't care" state to give significant compression of the tables over conventional two-state arrays. Typically, only n to n^2 words are necessary for functional memory instead of 2^{2n} words for conventional two-state arrays. In fact, such a functional memory has been suggested as a nucleus and the building block for the whole machine.

Lee, et al. [22] and Crane, et al. [23] have proposed a distributed logic memory structure which is suitable for LSI implementation. Although they suggested this structure for nonarithmetical logic operations, arithmetic can be performed by the bit serial table look-up method. But, this method is too slow when operated on scalar operands. However, for vector operands the arithmetic operation proceeds simultaneously in parallel on all components of the vector operands, and thus the inherent slowness of the bit serial table look-up method is masked by this parallelism. Bit serial processing is used in Goodyear's STARAN computer [24].

1.2.3 New number system representation - One of the main obstacles to the partition of currently existing arithmetic processors which use conventional binary representation (radix complement or sign magnitude) into identical subunits is the fact that the most significant digit behaves differently than the rest of the digit positions. Radix complement/diminished radix complement notation causes the control and structure of the most significant and least significant digits to be different from the rest of the digits due to such things as carry-in to the least significant digit, end-around carries and special circuitry for logical and arithmetical shifts. These factors preclude the chaining of accumulator modules to any desired length. Moreover, the radix or diminished radix complement notation causes problems both in the multiplication process (e.g., a correction factor) and in the division process. Sign magnitude notation is nice for multiplication and division because the sign of the result can be readily determined, but addition and subtraction need a complicated sequential control algorithm for determination of sign of the result. All this difficulty can be traced directly back to the limitations imposed by the requirement for knowledge of sign and magnitude of the operands and the result. This knowledge, by definition, is available on a word level rather than a digit level. So, it would be preferable to have a number representation where each digit position carries both magnitude and polarity information, unlike normal binary where the most significant bit carries sign but no magnitude information and other bits carry only magnitude but no sign information. This will remove the above limitation since a priori or a posteriori knowledge of

the operand or result magnitude and polarities is not necessary. This will make it possible to perform arithmetic on a digit ("stage" in a machine) basis rather than on a number ("register" in a machine) basis. That is, an arithmetic operation on corresponding digits of a pair (or more) of numbers would become invariant with respect to the polarities of two (or more) numbers in which they are separately imbedded. This results in two independent but important implications: One, that a true variable word length operation is completely practicable, permitting modular construction in terms of quantity of digit positions; and two, that simultaneous operations on multiple (two or more) operands are also practicable, permitting modular constructions in terms of number of operands.

The sign information with each digit position in a number can be provided either implicitly or explicitly. In a positional weighted number system, a negative radix implies [25] indirectly a sign associated with each digit position (positive for odd positions and negative for even positions, for integers). An example of explicit sign information with each digit is the Avizienis' signed digit number representation [26]. These two approaches can be utilized to design computational modules for each digit position, which can be used later on to perform arithmetic either in purely combinational logic net (array) or used with a sequential control algorithm. Shaipov [27] and Prangishvilli have proposed cellular arithmetic arrays using a basic computation module based on minus-two adder system. Avizienis and Tung [28],[29] have

proposed a universal arithmetic building element (ABE) to be used in combinatorial logic net to perform arithmetic functions. Pisterzi [30] has utilized the explicit signed-digit representation to design a limited connection arithmetic unit with a central global control which provides the temporally sequential commands to the various modules to achieve the arithmetic functions.

Negative base number system, while facilitating the addition/subtraction and multiplication processes at a digit level, makes the division process very complicated. In any restoring or non-restoring division algorithm [31],[32],[33], the signs of the partial remainder and the divisor are very essential and the negative base number representation does not lend itself for easy determination of sign of an operand because one has to go through a counting process to know whether the most significant digit of the integer representation is in an even position or an odd position. Further, for faster addition/subtraction, one still needs the carry look-ahead circuits.

Avizienis' proposed number representation, besides being signed-digit, is also redundant; i.e., each digital position can have more than r values where r is the radix of the number representation. This number system has many desirable features, namely,

i) The algebraic value z of the number Z composed of n + m + 1 digits $(z_{-n}\ldots z_{-1}\ z_0 \cdot z_1\ldots z_m)$ is given by the expression:

$$Z = \sum_{i=-n}^{m} z_i r^{-i}$$

ii) Algebraic value Z = 0 if and only if all $z_i = 0$.

iii) The sign of the algebraic value Z is given by the sign of the most significant (left-most) nonzero digit.

iv) To form the representation of the additive inverse -Z, the sign of every nonzero digit z_i is changed individually.

v) The addition and subtraction of two signed-digit operands Z and Y satisfy $s_i = f(z_i, y_i, z_{i+1}, y_{i+1})$ for all positions i, where s_i are digits in the representation of the sum or difference $S = Z \pm Y$. This means that there are no carry propagation chains in signed-digit additions (or subtractions).

vi) The same logic that is used for adding two numbers (maximally redundant) can be used to convert the number from conventional binary representation to the signed-digit format.

vii) It allows limited inspection of partial remainder digits to determine the quotient.

The properties (iv) to (vi) make the signed-digit redundant number representation very suitable for digit-wise operation of the arithmetic unit. Property (iii) obviates the need for any complement arithmetic operations.

Based on the above number representation, Avizienis [28] proposed his Arithmetic Building Element (ABE) which has the capability of adding two digits of the two operands to be added, forming the product of two digits and also of forming a sum of many digits, one of each different operand,

besides having the capability of achieving logical operations on individual digits. He proposed this element for use in combinational arrays for forming the product of two numbers. But since the ABE can form a sum of only $m \leq r+1$ digits, it becomes necessary to partition the product of two numbers where the multiplier is greater than r+1 digits long into groups of r+1 digits so that the same kind of ABE can be used to form the whole product. Secondly, the proposed ABE is too complex for any reasonable radix r. Thirdly, the combinational net for the division process is very complex and expensive.

For digit-wise arithmetic operations, mention should also be made of residue number representation [34] which allows addition/subtraction and multiplication on a digit basis. However, handling of overflow/underflow, conversion of conventional binary representation to residue number representation, and, of course, the division process are very complicated, and that is why not many computers have been built based on this number representation. Moreover, because the moduli for each digit position is different, all the digit modules are necessarily different and not compatible with fewer module type constraints of LSI.

1.3 Present Work

The goal of the present work is to formulate a set of desirable characteristics for an LSI implementable Arithmetic Unit capable of the four basic operations of Addition, Subtraction, Multiplication and Division, to choose a suitable system and logical organization which comes close

to meeting these desirable properties and finally to study the arithmetic and logic design of the arithmetic unit.

From our discussion in Sections 1.1 and 1.2, the following set of characteristics for the Arithmetic Unit are considered suitable for its implementation in LSI or any batch fabrication process technology.

i) The arithmetic unit should be partitionable on a bit slice or digit slice (for higher radix) basis which means that we should be able to perform calculations on a digit-by-digit basis. All the digit processing modules should be identical so that a variable word length can be accommodated.

ii) Purely combinational cellular arrays are too expensive for large operand lengths, especially when each cell is rather complex. Hence, the arithmetic function execution should be done by a time sequence of microinstructions. Further, to achieve a balance between the high cost of a purely combinatorial array and the slow speed of completely sequential execution of microinstructions, some form of pipeline structure should be employed so that when an arithmetic expression is evaluated, the various arithmetic operations can be overlapped.

iii) To avoid fan-out problems in case of large operand lengths, the various modules should have limited intercommunication with each other.

iv) Each processing module should have local control and be autonomous as far as possible so that only a few

microinstructions need to be issued by a central control to the modules, instead of a large number of separate control signals. This would cut down the number of external leads necessary on each module.

v) The various microinstructions should be as simple as possible.

vi) Each processing module must be consistent with the constraints of large scale integration insofar as total external pin count in the module is concerned and the module itself should preferably be made up of cells (identical logic repeated) when the cell consists of many many gates.

vii) Since the divide process by its very characteristic has to examine the most significant digits of the operands (dividend/partial remainder, and divisor) for the calculation of quotient, the multiplication and addition/subtraction should also be performed as a right-directed process.

This most significant digit first approach is consistent with other arithmethic processes of operand normalization, mantissa overflow determination and the determination of the sign of the result because these processes inherently require the examination of the most significant digits of the operands to determine what additional processing is necessary.

Many of the characteristics mentioned above are met by an Arithmetic Unit structure proposed by Pisterzi [30]. The Arithmetic Unit consists of modular processing elements called the Digit Processing Units (DPUs)[†] and a global control module called Primitive Control Unit (PCU).[†] The PCU

[†]DPU and PCU are the terminology used by Pisterzi [30]. In the present thesis, the terms PE and MCU will be used for the Processing Element and the global control module respectively.

does not broadcast control signals to all the DPUs but instead the PCU communicates only with the most significant DPU as far as the issuance of microinstructions is concerned. The first DPU executes each instruction and then passes it on to the second DPU which again executes this microinstruction and further passes it down to the next DPU and so on. Thus, a sort of pipeline of microinstructions is established where the same sequence of microinstructions is executed in each DPU.

A simplified block diagram of such an arithmetic unit is shown in Figure 1.1.

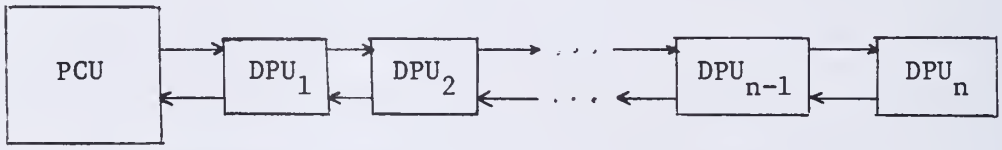


Figure 1.1 Block Diagram of a Basic Model of Limited Connection Arithmetic Unit

The present study concentrates on the design of the essential microinstructions necessary for performing four basic arithmetic operations in such a structure, the logic design of the Processing Element, the method of communication between the Processing Elements and the Data Main Memory for fetching and storing operands and results. The major part of this thesis reports on the logic design of the Processing Element and identifies those parts of the Processing Element whose gate and pin complexity are a function of the bit width of the Processing Element. This, in turn, allows us to choose a suitable bit width for the processing module consistent with the technology constraints and also to balance the costs for the processing logic and the control logic of the Processing Element.

Chapter 2 describes briefly the system and logical organization and mode of operation of the Arithmetic Unit. The major emphasis in this chapter is on the logical structure of the Mantissa Processing Logic (MPL), the method of communication between the modules of the MPL and the flow of microinstructions through them. This discussion provides the necessary perspective for the material in later chapters. The flow of microinstructions in the MPL is illustrated by a generalized example. The chapter concludes with the definition and a brief description of a set of basic and elementary microinstructions which are sufficient to execute the 'machine' arithmetic instruction like Add, Multiply of two operands.

Chapter 3 treats the arithmetic design of the Processing Element--the basic module of the Mantissa Processing Logic. The arithmetic design is described in terms of the implications of the particular structure of the Mantissa Processing Logic on the required characteristics of the number system, the number representation and the definition of a normalized number. Finally, in Section 3.6 which is the major portion of this chapter, we develop the definition and operational specification of a set of five simple arithmetic microinstructions. These microinstructions cause an arithmetic transformation of the data and are specified as such by an arithmetic transfer function, wherever possible. The digit algorithm for each arithmetic microinstruction is also given.

In Chapter 4, which is the largest chapter of this thesis, the logic design of the major components of the Processing Element is given. The major components are the register file for storage of active operands,

the Combinational Network for processing and the Control which generates control signals to condition the Combinational Network. This chapter also describes the actual format and code assignment for the twelve types of microinstructions executed in a Processing Element. Finally, the logic complexity of the Processing Element is calculated in terms of the total number of gates and external leads required in the Processing Element module as a function of the bit width of the module and the redundancy ratio of the multiplier and quotient digit.

Chapter 5 describes how the Mantissa Processing Logic and the Data Main Memory may communicate to fetch and store operands through an interface whose behavior is somewhat analogous to that of a cache memory.

In Chapter 6, we show how the various microinstructions can be combined into a sequence to be executed by the Processing Element modules to perform a 'machine' arithmetic instruction like Floating Point Add, Multiply, etc.

Summary and conclusions are given in Chapter 7.

Two appendices are included. Appendix A-1 gives the algebraic design of a digit recoder which changes the redundancy ratio of the digit from unity to $\leq 2/3$. In Appendix A-2, we calculate the number of radix-2^k digits of the truncated operands that are necessary in the model division to determine one radix-2^k quotient digit.

2. ORGANIZATION AND OPERATION OF ARITHMETIC UNIT

2.1 Introduction

In order to put the discussion in the following chapters in proper perspective, a brief description of the logical organization and method of performing the processing is given. The method of processing is illustrated by an idealized example in Section 2.4. The chapter closes with an introductory description of the repertoire of only the essential microinstructions which are executed by the processing logic.

2.2 Organization of the Arithmetic Unit

In Figure 2.1 is shown the global block diagram of the arithmetic unit. It consists of Mantissa Processing Logic, Exponent Processing Logic, Local Operand Memories (LOMM, LOEM) and an Arithmetic Control Unit. The Arithmetic Control Unit (ACU) consists of three parts--the Global Arithmetic Control Unit (GACU), the Mantissa Control Unit (MCU) and an Exponent Control Unit (ECU).

The GACU acts as the interface between the arithmetic unit and the rest of the computer. It receives the arithmetic instructions from the central control of the computer, decodes them, and causes the Local Operand Memory Control (LOMCO) to fetch the ncessary operands from main memory, if they are not already present in LOMM. LOMCO provides the LOMM address of the operands to the GACU which then issues the necessary commands to the ECU and MCU for exponent and Mantissa processing and coordinates their actions. After the processing is complete, it informs

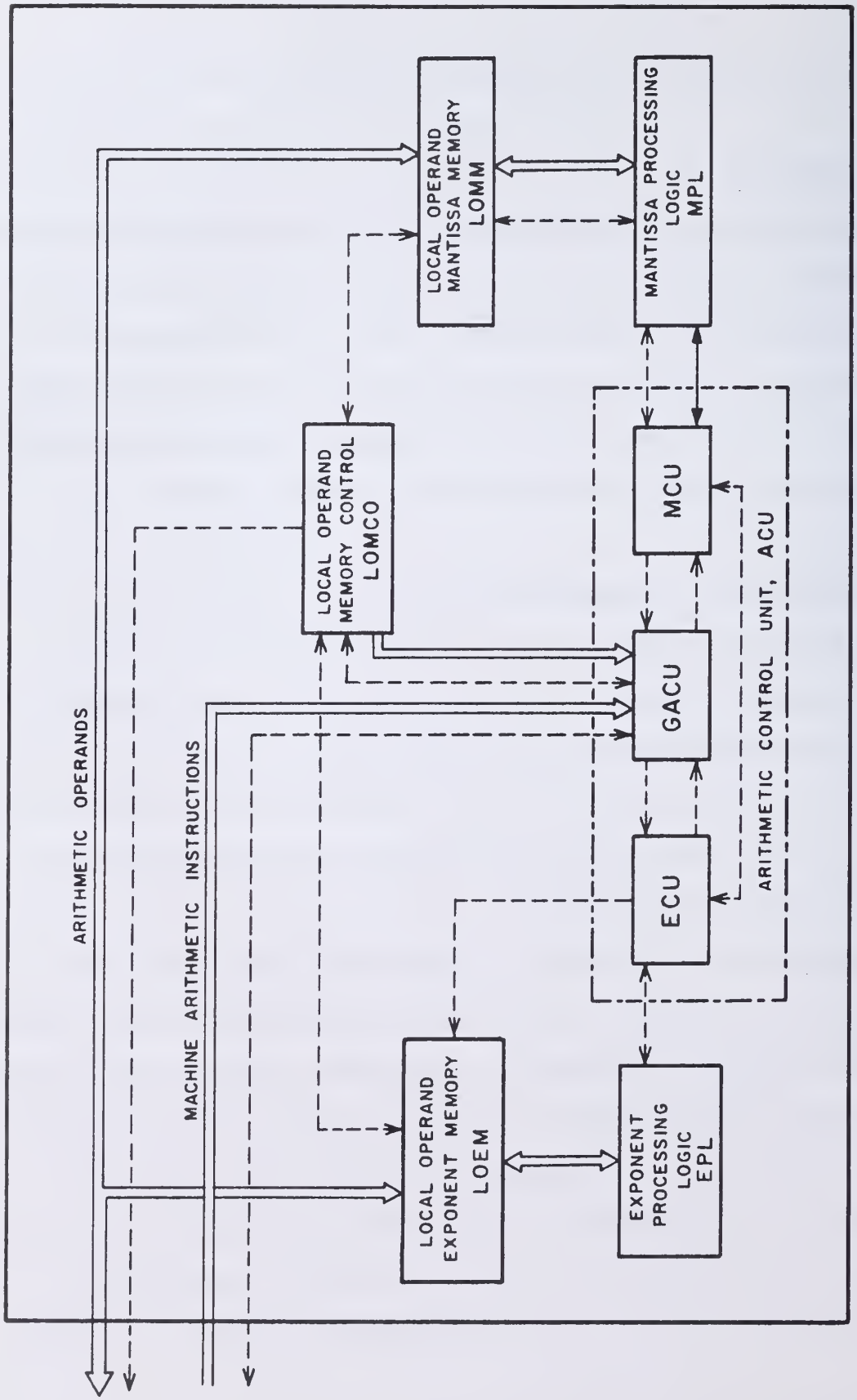


Figure 2.1 Global Block Diagram of Arithmetic Unit.

the central control its status along with any exceptional conditions if necessary, that may arise during execution of the instruction.

The MCU converts the commands received from the GACU into necessary microinstructions to be executed by the Mantissa Processing Logic. For example, the Multiply command is converted into a series of shift left multiplier, form multiple and add, and shift left accumulator. Also, it contains the overflow recoder logic and quotient determination logic, etc.

The ECU performs the necessary control for exponent arithmetic such as calculating the difference of the exponents for addition and subtraction arithmetic instruction, sum of the exponents for the multiplication instruction and detecting exponent overflow and underflow conditions.

In this thesis, we shall be concerned mainly with the detailed design of the Mantissa Processing Logic and its communication with the Local Operand Mantissa memories. The detailed design of GACU, MCU and ECU is beyond the scope of this research. The next section describes the logical organization of the Mantissa Processing Logic and a description of the method of processing.

2.3 Organization of Mantissa Processing Logic

The Mantissa Processing Logic consists of a linear cascade of identical Processing Elements (PEs). Each PE is a complex logical module and contains logic to perform the various microinstructions, issued by the Mantissa Control Unit (MCU), in cooperation with other PEs. The MCU communicates only with the most significant PE (closest to the

MCU) and the microinstructions flow serially (in a pipelined manner) from the most significant PE to the least significant PE.

Figure 2.2 shows the schematic organization of the Mantissa Processing Logic along with the MCU. This figure also shows an End Unit which is optional and not intrinsically necessary for the arithmetic processing. The End Unit allows the last PE to be identical to all the other PEs as far as interface is concerned, thus causing it to operate as though it had another PE to its right. Moreover, it could contain some logic in which the operand digits shifted off the right end could be temporarily stored for improving the accuracy of the result [35].

The PEs collectively contain the fractional (Mantissa) parts of all active operands, one digit in each PE, as shown in Figure 2.3. Because the quotient generation and operand normalization processes require the examination of most significant digits, the operands are placed in the PEs so that the digits of each of the operands are available to the microinstructions in order of decreasing significance. Thus, the most significant digits of the active operands are placed in the PE which communicates with the MCU.

Each PE performs the same sequence of microinstructions. A given microinstruction is not executed by all PEs in synchro-parallelism but rather must be executed by them in sequence (i.e., first by PE_1, then PE_2,...). Note that this is different from a conventional pipeline organization in which data flows in sequence through a number of stages which, in general, do different operation on the data. In this organization, however, data is relatively constant and flowing microinstructions

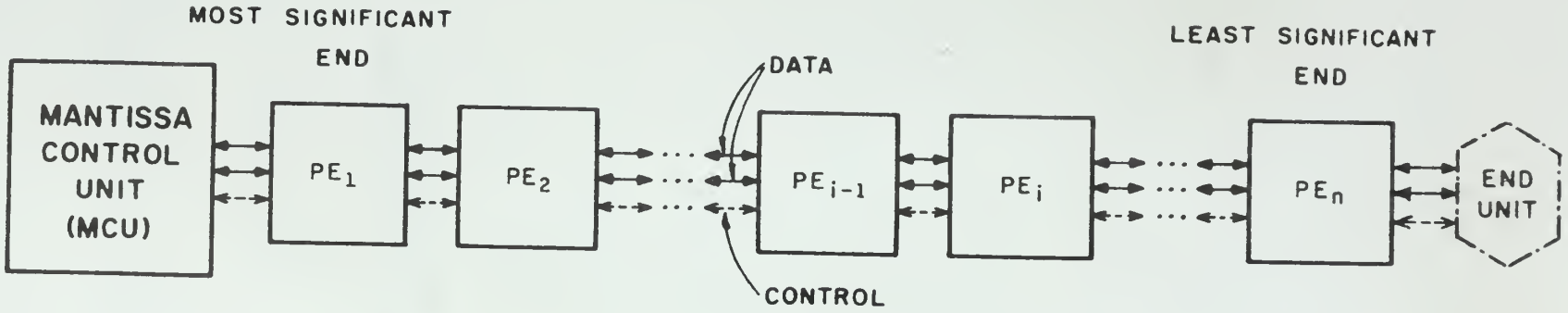


Figure 2.2 The Organization of the Limited Connection Mantissa Processing Logic.

	PE_1	PE_2	PE_3	. . .	PE_n
A operand	a_1	a_2	a_3	. . .	a_n
M operand	m_1	m_2	m_3	. . .	m_n
ϕ operand	ϕ_1	ϕ_2	ϕ_3	. . .	ϕ_n
.	.	.	.	. . .	.
.	.	.	.	. . .	.
.	.	.	.	. . .	.
Z operand	z_1	z_2	z_3	. . .	z_n

$$A = \sum_{i=1}^{n} a_i \, r^{-i}$$

$$M = \sum_{i=1}^{n} m_i \, r^{-i}$$

etc.

where r is the radix.

Figure 2.3 The Distribution of Operands Digits in the PEs of Mantissa Processing Logic.

tell a PE what operation to execute on the data resident in that PE at that instant of time.

During processing, each PE physically communicates only with its immediate neighbors. To execute a microinstruction, a given PE may need information from its right neighbor. This information logically may depend on the contents (active operand digits) of its neighboring PEs, depending on the nature of the microinstruction. So we may say that each PE physically communicates with only one PE to its immediate right but from a logical viewpoint, the PE communicates with more than one PE.† In the following discussion of the mode of processing, when we talk about information required by a PE, from its right neighbors, we mean the information requirement in the logical sense.

As mentioned earlier, a given microinstruction is executed by PEs not in synchro-parallelism but rather in sequence. As soon as all the PEs (say α_j) which contain information required by PE_1 to perform microinstruction j+1 (referred to as μ_{j+1}) have executed μ_j and have sent the required information to PE_1, μ_{j+1} may be performed by PE_1. The microinstructions, executed by PEs, are defined in such a way that they have regular data requirements independent of the position of the PE in which a microinstruction is executed so that as each additional PE executes μ_j, one more PE may execute μ_{j+1}.†† The microinstructions may be viewed as

---

†The logical communication could be converted into physical communication by duplicating the necessary hardware logic in the PE where that information is required but this would increase the number of interconnections.

††There is an exception to this rule in the case of Assimilation Recode (AR) microinstruction in which case the α_j is variable and depends on the nature of the data resident in the PEs. This is further explained later in Section 4.3.2.3.3.

flowing through successive PEs. Clearly, the PE registers do not contain entire operands as long as any of the PEs are actively executing microinstructions. Each PE contains the digits from the results of the last microinstruction executed. (In the worst case, if there are n PEs and each PE has the capability of storing n active operands, there could be n active operands in different stages of processing if there is a sequence of n load or store microinstructions.)

2.4 Formal Description of Processing in a PE

The processing performed by the PEs can be described by the following:

Let

$$ {}_{j}\vec{X}_{i} = \psi_{j} \left({}_{j-1}\vec{X}_{i},\ {}_{j}F_{i-1},\ {}_{j}G_{i+1} \right) \tag{2.1} $$

$$ {}_{j}F_{i} = \phi_{j} \left({}_{j-1}\vec{X}_{i},\ {}_{j}F_{i-1} \right) \quad \text{and} \tag{2.2} $$

$$ {}_{j}G_{k} = \Gamma_{j} \left({}_{j}F_{k-1},\ {}_{j-1}\vec{X}_{k},\ {}_{j-1}\vec{X}_{k+1}, \ldots, {}_{j-1}\vec{X}_{k+\alpha_{j}} \right) \tag{2.3} $$

where

$ {}_{j}\vec{X}_{i} $ is the operand information contained in the i-th PE immediately following the execution of μ_{j}. It consists of the i-th digit of each of the active operands. μ_{j} represents the j-th microinstruction,

ψ_{j} is the function employed to obtain the new operand set and is dependent on the microinstruction to be performed,

$ {}_{j}F_{i} $ is a 'modifier' value which PE_{i} transmits to PE_{i+1} with the microinstruction j, to be performed next,

ϕ_j is the function which each PE performs to determine ${}_jF_i$,

Γ_j is the function PE_k employs to determine ${}_jG_k$,

${}_jG_k$ is the value which PE_k transmits to the PE executing μ_j, and

α_j is one more than the number of PEs which must logically cooperate with the right neighbor of PE performing μ_j in order to generate the necessary ${}_jG_{i+1}$.

The information ${}_jG_k$ is generated in a time sequential fashion. ${}_jG_k$ consists of α_j components ${}_jG_k^0, {}_jG_k^1, \ldots, {}_jG_k^{\alpha_j-1}$ and they are given by the following relations.

$$
\begin{aligned}
{}_jG_k^0 &= \Gamma_j^0 ({}_jF_{k-1}, {}_{j-1}\vec{X}_k) \\
{}_jG_k^1 &= \Gamma_j^1 ({}_jF_{k-1}, {}_{j-1}\vec{X}_k, {}_jG_{k+1}^0) ; \\
&\vdots \\
{}_jG_k^{\alpha_j-1} &= \Gamma_j^{\alpha_j-1} ({}_jF_{k-1}, {}_{j-1}\vec{X}_k, {}_jG_{k+1}^0, \ldots, {}_jG_{k+1}^{\alpha_j-2}) \\
{}_jG_k &= \{{}_jG_k^0, {}_jG_k^1, \ldots, {}_jG_k^{\alpha_j-1}\}
\end{aligned}
\qquad (2.4)
$$

The superscript on ${}_jG_k$ indicates the time order of sequential generation of G-information.

Another formulation which is applicable for only fixed value of α_j is given by Pisterzi [30]. In this formulation, the PE executing microinstruction μ_j gets G-information directly from α_j PEs to its immediate right. The trade-off between the two is that the former needs less con-

nections to PE_i and also less logic in PE_i since the G-information is developed in a distributed fashion in the α_j PEs. However, this is obtained at the expense of more complex control and longer time delay.

The operation of a typical PE, PE_i say, is as follows. It begins in a state in which it is receptive to information defining the next microinstruction to be performed. PE_i receives this information (microinstruction) and the value of ${}_jF_{i-1}$ from its left neighbor PE_{i-1}. Then PE_i determines ${}_jG_i$--the information required by PE_{i-1} to complete microinstruction μ_j. ${}_jG_i$ is determined sequentially as described by the set of relations in (2.4). The component ${}_jG_i^0$ is developed immediately. At the same time, PE_i determines ${}_jF_i$ by performing equation (2.2)(which incidentally is the same as ${}_jF_{i-1}$ in <u>most</u> cases) and transmits the identity of μ_j along with ${}_jF_i$ to PE_{i+1}. At this time, PE_{i+1} generates ${}_jG_{i+1}^0$ and transmits it back to PE_i so that PE_i may generate ${}_jG_i^1$. Simultaneously, it (PE_{i+1}) transmits the identity of μ_j instruction along with ${}_jF_{i+1}$ to PE_{i+2} which repeats the same process. Note that the information ${}_jG_i^{\alpha_j-1}$ depends on ${}_jG_{i+\alpha_j}^0$ which must trickle back to PE_i. Although this takes quite some time, the ${}_jG_{i+1}^{\alpha_j-1}$ can be generated by PE_{i+1} just one time step later. Initial setup time is large, however,

As soon as PE_i transmits ${}_jG_i^{\alpha_j}$ to PE_{i-1}, PE_{i-1} can complete the execution of microinstruction μ_j. After some time, PE_i receives a signal from PE_{i-1} indicating that PE_{i-1} has executed μ_j. PE_i then executes μ_j (the necessary ${}_jG_{i+1}^{\alpha_j}$ being ready by now). PE_i now transmits a signal to PE_{i+1} which indicates that PE_{i+1} may execute μ_j. When PE_i receives an acknowledgement from PE_{i+1}, it goes into a state where it is receptive to information concerning μ_{j+1}. The sequence above then repeats.

2.5 Generalized Example

To illustrate how the processing of several microinstructions may take place concurrently in the Mantissa Processing Logic, each by a different PE, we describe below a generalized example. This example is borrowed from Pisterzi [30] but the necessary changes have been made to conform to our notation.

Table 2.1 shows the α_j for the various microinstructions for the generalized example. The Mantissa Processing Logic will have five PEs and one operand. This operand will be indicated as composed of five digits ${}_ja_1,\ldots,{}_ja_5$ such that digit ${}_ja_i$ is the digit contained in PE_i after the j-th microinstruction.

Table 2.1

α_j of the Microinstructions of the Example of Figure 2.4.

j	1	2	3	4	5	6
α_j	2	1	0	1	2	0

The operation of the Mantissa Processing Logic is presented in a tabular form in Figure 2.4. The columns labeled O_i will indicate the operand contained in PE_i. The occurrence of ${}_ja_i$ in the i-th operand column will indicate that ${}_ja_i$ has just been computed and placed in the operand register of PE_i. The columns labeled IR_i will indicate the microinstruction being executed by PE_i and/or the G-information being produced by PE_i. The occurrence of μ_j in the IR_i column will be used to

	MICROINSTRUCTION					OPERAND REGISTER				
	IR_1	IR_2	IR_3	IR_4	IR_5	O_1	O_2	O_3	O_4	O_5
1	$\mu_1, {}_1G_1^0$					0^a1	0^a2	0^a3	0^a4	0^a5
2		$\mu_1, {}_1G_2^0$								
3	${}_1G_1^1$		$\mu_1, {}_1G_3^0$							
4		${}_1G_2^1$		$\mu_1, {}_1G_4^0$						
5	(μ_1)		${}_1G_3^1$		$\mu_1, {}_1G_5^0$	1^a1				
6	$\mu_2, {}_2G_1^0$	(μ_1)		${}_1G_4^1$			1^a2			
7		$\mu_2, {}_2G_2^0$	(μ_1)		${}_1G_5^1$			1^a3		
8	(μ_2)		$\mu_2, {}_2G_3^0$	(μ_1)		2^a1			1^a4	
9	(μ_3)	(μ_2)		$\mu_2, {}_2G_4^0$	(μ_1)	3^a1	2^a2			1^a5
10	$\mu_4, {}_4G_1^0$	(μ_3)	(μ_2)		$\mu_2, {}_2G_5^0$		3^a2	2^a3		
11		$\mu_4, {}_4G_2^0$	(μ_3)	(μ_2)				3^a3	2^a4	
12	(μ_4)		$\mu_4, {}_4G_3^0$	(μ_3)	(μ_2)	4^a1			3^a4	2^a5
13	$\mu_5, {}_5G_1^0$	(μ_4)		$\mu_4, {}_4G_4^0$	(μ_3)		4^a2			3^a5
14		$\mu_5, {}_5G_2^0$	(μ_4)		$\mu_4, {}_4G_5^0$			4^a3		
15	${}_5G_1^1$		$\mu_5, {}_5G_3^0$	(μ_4)					4^a4	
16		${}_5G_2^1$		$\mu_5, {}_5G_4^0$	(μ_4)					4^a5
17	(μ_5)		${}_5G_3^1$		$\mu_5, {}_5G_5^0$	5^a1				
18	(μ_6)	(μ_5)		${}_5G_4^1$		6^a1	5^a2			
19		(μ_6)	(μ_5)		${}_5G_5^1$		6^a2	5^a3		
20			(μ_6)	(μ_5)				6^a3	5^a4	

Figure 2.4 Illustration of the Execution of the Generalized Example in Mantissa Processing Logic.

denote that PE_i has just received the identity of j-th microinstruction and will begin determining ${}_jG_i$ in a time sequential fashion. The appearance of ${}_jG_i^{\lambda}$ in the IR_i column indicates that PE_i has just determined the λ-th component of ${}_jG_i$ information which is needed by PE_{i-1}. λ ranges from 0 to α_j-1. (In our example, $0 \leq \lambda \leq 1$.) The occurrence of (μ_j) will represent that the execution of microinstruction μ_j has just been completed by PE_i (and the result operand digit ${}_ja_i$ has been generated, as indicated by the appearance of ${}_ja_i$ in column O_i). The progression of time will be indicated by the rows, each row equivalent to the time required by a PE to execute one step of processing.

Figure 2.4 shows the Mantissa Processing Logic in steady state at time 0. No microinstructions are being executed and the operand A_0 $({}_0a_1, {}_0a_2, {}_0a_3, {}_0a_4, {}_0a_5)$ is in the operand register. The processing proceeds as follows.

We assume that at time 3, the identity μ_1 of microinstruction 1 has reached PE_3. PE_3 calculates ${}_1G_3^0$ and sends it to PE_2 (Figure 2.5a). At time 4, PE_2 calculates ${}_1G_2^1$ and sends it to PE_1. At time 5, all the G-information required for execution of μ_1 (α_1=2) is available in PE_1 and μ_1 is executed by PE_1. This causes ${}_0a_1$ to be replaced by ${}_1a_1$. During the next four time intervals, μ_1 is performed consecutively by each of the remaining PEs since ${}_1G_k^1$ becomes available just as it is required by PE_{k-1} to perform μ_1. The identity μ_2 of the second microinstruction is received by a PE, one time unit after that PE performs μ_1. Since PE_1 requires ${}_2G_2^0$ to execute μ_2 (α_2=1), this microinstruction is not performed by PE_1 until time 8, one time step after PE_2 is able to determine this value and send it to PE_1 (Figure 2.5b). Just as with μ_1, μ_2 is executed

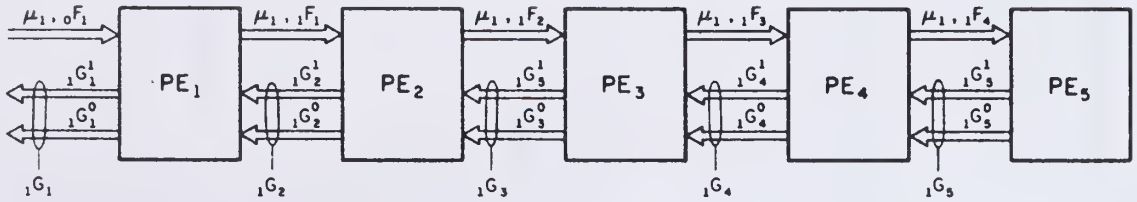


Figure 2.5a Illustration of Processing for Microinstruction μ_1 and $\alpha_1 = 2$.

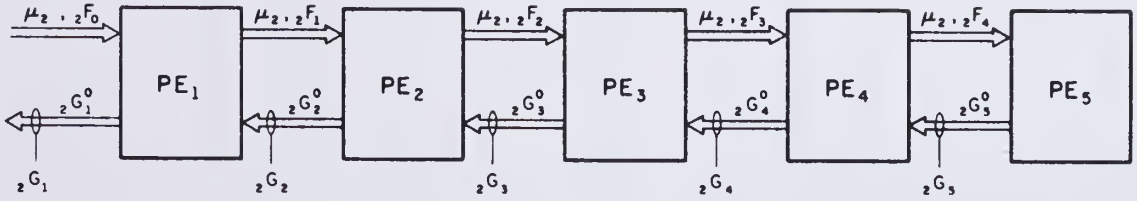


Figure 2.5b Illustration of Processing for Microinstruction μ_2 and $\alpha_2 = 1$.

sequentially by each of the remaining PEs during each of the next four time intervals. Microinstruction μ_3 is performed by each of the PEs one time unit after each PE has performed μ_2 because $\alpha_3 = 0$ and no outside information is required. The other microinstructions are performed in the same pattern.

In general, PE_i performs μ_j, $2\alpha_j + 1$ time units later following the execution of μ_{j-1}. The time $T_{\Sigma m}$ elapsed between the instant when the identity of the first microinstruction reaches PE_1 and the instant of execution of the m-th microinstruction (of a set of consecutively issued microinstructions) by the first PE_1 is given by

$$T_{\Sigma m} = \sum_{j=1}^{m} 2\alpha_j + m.$$

2.6 The Micro-Instruction Repertoire of the PEs

In this section, we will discuss briefly the microinstructions which are executed by the PEs so that the overall arithmetic unit is able to do addition, subtraction, multiplication, division and normalization. The microinstructions may be broadly categorized in five classes for the purposes of this discussion. These five classes are:

1. the inter-register transfers,
2. the shift microinstructions,
3. the arithmetic microinstructions,
4. the memory accessing microinstructions, and
5. the miscellaneous microinstructions.

2.6.1 The inter-register transfer microinstructions - These microinstructions cause operands to be transferred from one internal register of the PE to another internal register. There are two instructions in this class: Transfer Direct (TD) and Transfer Invert (TI). The microinstruction TD moves the contents of one register in the PE to another register, both the registers being specified explicitly in the microinstruction, with no changes in the source operand. The microinstruction, TI, on the other hand, causes the transfer of operands from source to destination register, with the sign of the source operand being inverted, that is, changed to opposite polarity.

The microinstruction TD allows the results of one instruction to be stored temporarily into another local register before being used as an operand in the execution of some later microinstruction, thus avoiding a memory reference. A second application of this microinstruction is in the exchange of operands when normalization is required. As would be seen later on, because the Normalization Recode (NR) and Assimilation Recode (AR) microinstructions require the operand to be in only the Accumulator register, assimilation and normalization of operands would require the use of microinstruction TD for moving the operand to the Accumulator register.

The main use of the microinstruction TI occurs when one needs to change the sign of an operand before being used, e.g., in the case of subtraction. Since the PE has only an 'add' microinstruction, it is necessary to invert the sign of the operand before being 'added' to another operand to cause subtraction. Note that, in this microinstruction,

the source and destination register addresses can be the same. This microinstruction can thus be used, if necessary, for getting the absolute value of an operand.

In all the inter-register transfers, all of the data required by a PE to perform the microinstruction is contained within that PE itself. It can be seen in Figure 2.3. Each PE contains one digit of each of the operands. Therefore the value of α, the number of PEs which must logically cooperate with the PE executing the inter-register transfer microinstruction, is zero, and ${}_jF_i$ is not required to transmit data. The value of ${}_jF_i$ is used instead to identify both the registers taking part in the transfer. The exact format of the microinstructions TD and TI is discussed in Section 4.3.2.2.

In the notation of Section 2.4, the inter-register transfer microinstructions may be expressed as:

$$ {}_jX_i = {}_{j-1}Y_i \qquad i = 1, 2, \ldots, n \qquad (2.5) $$

$$ {}_jF_i = {}_jF_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.6) $$

$$ {}_jG_i = \langle\text{null}\rangle \qquad i = 1, 2, \ldots, n \qquad (2.7) $$

where

Y is the register to be copied into the X register,

${}_jX_i$ is the i^{th} digit of the X register after the transfer,

${}_{j-1}Y_i$ is the i^{th} digit of the Y register before the transfer, and

<null> indicates that the value of ${}_jG_i$ is not required when performing inter-register transfers.

2.6.2 The shift microinstructions - These microinstructions are used during radix point alignment prior to addition or subtraction, for normalization, and for multiplication and division by the radix during the repetitive steps for multiplication and division. A shift of more than one digital position is performed as a number of successive shifts of one digital position each.

The left shift can be accomplished by causing the PE to the immediate right of the PE performing the microinstruction to transmit the value of the digit of the operand contained in its register to the PE performing the microinstruction. This PE stores the digit it receives in its operand register. The equations defining the left shift microinstruction, LS, are:

$$ {}_{j}x_{i} = {}_{j}G_{i+1} \qquad i = 1, 2, \ldots, n \qquad (2.8) $$

$$ {}_{j}F_{i} = {}_{j}F_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.9) $$

$$ {}_{j}G_{i} = {}_{j-1}x_{i} \qquad i = 1, 2, \ldots, n \qquad (2.10) $$

$$ {}_{j}G_{n+1} = {}_{j}F_{n} \qquad \text{if } {}_{j}F_{n} \text{ is a valid digit, otherwise see text} \qquad (2.11) $$

where

X is the operand being shifted,

${}_{j}x_{i}$ is the i^{th} digit of the shifted operand,

${}_{j-1}x_{i}$ is the i^{th} digit of X before the shift,

${}_{j}F_{i}$ is the modifier value passed along with the microinstruction and carries the address of the register to be shifted and the value of a digit sent by MCU that is to go into the last PE. This is made use of in the execution of Multiplication.

${}_jF_0$ is the value that the MCU sends to PE_1 with the left shift microinstruction to indicate the value that is to go into the last PE. If ${}_jF_0$ is a valid digit, it becomes the digit shifted into the last PE. If it is not a valid digit, it causes the End Unit to shift-in the digit shifted out during the last right shift.

One should also note that the left shift microinstructions make it possible to transmit the most significant digit of an operand to the MCU. The left shift can therefore be used by the MCU to examine operands.

The right shift (RS) microinstruction does not have the complexity of the left shift microinstruction. The value stored into a PE is the value transmitted by its left neighbor PE with the indication that a right shift is to be performed. The value of the digit to be stored in the first PE is determined by the MCU. In the terminology of Equations 2.1 through 2.3,

$$ {}_jx_i = {}_jF_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.12) $$

$$ {}_jF_i = {}_{j-1}x_i \qquad i = 1, 2, \ldots, n \qquad (2.13) $$

$$ {}_jG_i = \text{<null>} \qquad i = 1, 2, \ldots, n \qquad (2.14) $$

where

${}_jF_0$ is the digit which the MCU transmits with the indication that a right shift is to be performed. This value becomes the value of the most significant digit of the shifted operand.

The value of ${}_jF_n$, which is transmitted by PE_n to the 'End Unit', is stored as the new top element in the push-down stack. The push-down stack is essentially an extended version of 'guard' digits.

A final note concerning shifts is that the value of $\alpha_{LS} = 1$ and $\alpha_{RS} = 0$. The exact format of the microinstructions LS and RS is described in Section 4.3.2.2.

2.6.3 The arithmetic microinstructions - The microinstructions in this class are those instructions which do some sort of arithmetic transformation on the operands. These microinstructions operate on one, two or more than two operands, depending on the nature of the micro-instructions. The various microinstructions in this class are: Form Multiple and Add (FMA), Simple Sum (SS), Multiple Sum (MS), Assimilation Recode (AR) and Normalization Recode (NR).

The microinstruction FMA is used to form the product of a multiplier (quotient) digit and a multiplicand (divisor) digit and add (subtract) it to (from) the partial product (partial remainder) in the execution of Multiplication (Division) instruction and is the most complex of all microinstructions.

The microinstruction, SS, sums the contents of two registers and is used to execute the Add or Subtract instructions. Although the micro-instruction FMA could be used for this purpose, a separate microinstruction SS was designed for faster operation, especially because the frequency of addition or subtraction of two operands in a computer program is much higher than multiplication or division.

The Multiple Sum microinstruction, MS, is used to add the contents of more than two registers in a PE. This microinstruction is not intrinsically necessary for the operation of the arithmetic unit but rather comes about as a useful by-product of the design of the logic for microinstruction FMA.

The microinstruction NR operates on a single operand in the Accumulator register only. It is used to recode the operand in a form which when left-shifted one or more places meets the normalization definition.

Finally, the Assimilation Recode microinstruction, AR, is used to convert the operand in the Accumulator register, from the number representation used in the arithmetic processing, to the conventional form for communication to memory or other parts of the computer system. This microinstruction is very similar to microinstruction NR.

All the above microinstructions are discussed in detail in Chapter 3.

2.6.4 The memory-accessing microinstructions - These microinstructions cause the exchange of data between the internal registers of the PEs and a local buffer operand memory. They are used to fetch operands into PEs for processing and to store the results for eventual transmission to the main memory of the computer. The two microinstructions are Load from Processor Memory (LPM) and Store into Processor Memory (SPM); the former is used to bring operands into PE registers and the latter causes the contents of a specified PE register to be stored into a specified location of the local Operand Processor Memory. The

microinstructions in this class are similar to the inter-register transfer microinstructions except that one of the source or destination address refers to some location in the local Operand Processor Memory. In these microinstructions also, the modifier ${}_jF_i$ is used to identify the source and the destination. The exact format of these microinstructions are discussed in Section 4.3.2.2. The communication of the PEs with the Data Main Memory via the local Operand Processor Memory is discussed in Chapter 5.

2.6.5 The miscellaneous microinstructions - One instruction in this class is Load Constant (LDC). This microinstruction can be used to clear the operand register by loading zeros in a specified register of all the PEs in the arithmetic unit. It can also be used to initialize an operand register spread across all the PEs to a pattern such that all the digits are identical. An example of such an use could be the loading of maximum value of operands. In the terminology of Section 2.4,

$$ {}_jX_i = {}_jF_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.15) $$

$$ {}_jF_i = {}_jF_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.16) $$

$$ {}_jG_i = \text{null} \qquad i = 1, 2, \ldots, n \qquad (2.17) $$

${}_jF_o$ = digit which the MCU sends to PE_1 with the LDC microinstruction and the register name in which the constant is to be loaded.

Clearly, $\alpha_{LDC} = 0$ which means that no information is needed from its right neighbor for the execution of this microinstruction. Note that in Equation 2.15, only the digit part of field ${}_jF_{i-1}$ is stored in register ${}_jX_i$.

3. ARITHMETIC DESIGN AND IMPLEMENTATION CONSIDERATIONS

3.1 Introduction

This chapter describes the arithmetic design of the Processing Element. Arithmetic Design consists of the choice of a suitable number system, number representation, and the development of suitable digit level algorithms. Serial processing in an iterative structure has important implications on all of these factors and will be considered in this chapter. Implementation of the digit algorithm and its implications for LSI realization of the Processing Element are also discussed.

3.2 Implications of Serial Processing on Arithmetic Design

From the description of processing in Section 2.4, it is evident that the results are obtained on a digit-by-digit basis. To achieve a compromise between the digit serial processing and the arithmetic speed, the arithmetic should be carried out in higher radix say $r = 2^k$ $(k > 1)$ such that k bits of the result are obtained at any step.

Since the processes of quotient generation, operand normalization, mantissa overflow determination and the determination of the sign of the result inherently require the examination of the most significant digits of operands to determine what additional processing is necessary, arithmetic algorithms should be so designed that the most significant digits of the result are obtained first. The most-significant-digit-first (MSDF) approach has the advantages of providing early status indication (overflow, sign of the result, etc.), normalization concurrent with processing

and early termination of processing as soon as enough significant digits in the result have been obtained. The latter would allow faster variable precision arithmetic in a digit serial environment. Early status indication would also aid in an instruction look-ahead unit. Further, the MSDF approach allows the meshing-in (pipeline) of successive macroinstructions for efficient operation. For example, if a MULTIPLY instruction is followed by a DIVIDE instruction, at some point in time, the least significant digits of the product can be generated by a right directed procedure in the least significant elements of the iterative structure, while the most significant elements are generating quotient digits.

3.3 Choice of Number System

For a smooth flow of microinstructions in the linear iterative structure and for maximizing the rate of computation, two constraints on α_j[†] are necessary:

a) The microinstructions should have regular data requirements independent of the significance of the digits retained by a PE. That is, α_j should be constant.

b) The value of α_j should be as small as possible because the execution rate of a given microinstruction is inversely proportional to α_j.

In a conventional weighted number system, a carry or borrow into any digital position is a function of all the digits to the right of this position.

[†] α_j is the number of PEs from which a given PE requires information (in the logical sense) in order to execute the microinstruction μ_j.

Thus for MSDF algorithms which are right directed, the conventional number system cannot be employed because in a conventional number system, the value of α_j is a function of the significance of the digit itself. A redundant number system which gives a bounded value of α_j is clearly essential.

3.4 Choice of Number Representation and Amount of Redundancy

The major factors influencing the choice of the redundant number representation and the amount of redundancy in the number system are the following:

a) the ease of conversion from the conventional number representation to the redundant number representation,

b) its compatibility with the widely employed conventional binary number system,

c) ease of normalization of operands to radix-2 limits, and

d) LSI technology constraints, namely

   (i) minimization of the number of types of cells (in the arithmetic and logic sense) required for higher radix ($r = 2^k$) implementation of the digit processing logic, and

   (ii) minimization of the number of input and output pins.

In this study, signed-digit redundant number representations with maximal redundancy were chosen, because they satisfy most of these requirements.

3.4.1 Signed-digit number representations

Signed-Digit (SD) representations are redundant positional representations.

A number X is represented, in radix-r, redundant, signed-digit format, as a digit vector (abbreviated as "d-vector") of length n + m + 1

$$X = x_{-m}\, x_{-(m-1)} \cdots x_0\, x_1\, x_2 \cdots x_{n-1}\, x_n$$

such that

$$X = \sum_{i=-m}^{n} x_i\, r^{-i}$$

where

$$x_i \in \{\bar{d},(\bar{d}-1),\ldots,\bar{1},0,1,\ldots,(d-1),d\}$$

and

$$\left\lceil \frac{r}{2} \right\rceil \leq d \leq (r-1).$$

The overbar indicates negative values and unless otherwise specified, we shall be using rightward indexing in the d-vector representation. For maximally redundant, signed-digit number systems

$$d = r - 1.$$

That is, for a radix $r = 2^k$, each digit of the radix-r digit vector can assume any integer value in the digit set $\{\overline{(2^k-1)},\ldots,\bar{1},0,1,\ldots,(2^k-1)\}$. Some of the desirable properties of signed-digit representations are:

1. Representation of zero is unique. An algebraic value of X = 0 if, and only if, all $x_i = 0$.
2. The additive inverse (negation) of an operand is very simply achieved by reversing the sign of every non-zero digit individually.
3. The sign of the algebraic value of X is given by the sign of the most-significant (leftmost) non-zero digit.

4. For the sum or difference of two signed-digit operands,

$\alpha_j = 1.$

Maximal redundancy is compatible with the widely used sign-magnitude representation of conventional binary input operands. A binary number may be interpreted as a number of radix $r = 2^k$ by grouping the binary digits into groups of k bits each. Conversion from the conventional number system to signed-digit form is simply carried out by just attaching the sign of the conventional number to each digit. Another important advantage is the fact that the carry between bits of a digit has the same properties as the carry between digits whereas in the other than maximally redundant representations such is not the case. This allows the radix-2 arithmetic, for example shifts, etc., if necessary. From the LSI viewpoint, it allows a radix $r = 2^k$ arithmetic structure to be composed of k identical and simpler radix-2 substructures interconnected in a regular pattern. Maximal redundancy also provides more code-space patterns [36] for testing the radix-r module. This makes the design of a self-testing version of the module easier.

Two modes of representation for a signed-digit of the radix-2 d-vector are used, depending on the area of application:

a) Sign-Magnitude (SM_r) Mode - Each radix-2^k digit x_i is represented by a single sign bit s_i and k magnitude bits, x_{i_j} (j=0,1,...,k-1) such that

$$x_i = (-1)^{s_i} \sum_{j=0}^{k-1} x_{i_j} \cdot 2^j \quad , \quad s_i, x_{i_j} \in \{0,1\}$$

b) Redundant-Binary (RB_r) Mode - Each radix-2^k digit x_i is represented by k redundant binary digits $x^*_{i_j}$ (j=0,1,...,k-1), such that

$$x_i = \sum_{j=0}^{k-1} x^*_{i_j} \, 2^j \quad , \quad x^*_{i_j} \in \{\bar{1},0,1\}.$$

(Note that in the above representation of x_i in terms of radix-2 sub-digits, we use zero-origin leftward indexing.)

The SM_r mode requires k+1 binary storage elements (or k+1 pins as an output from the processing element) and the RB_r mode needs 2k binary storage elements (or pins) because each redundant binary digit requires two binary state elements. The SM_r representation for a radix-r digit is used for inter-PE communication to keep the number of external I/O pins small. The RB_r mode of representation is used for implementing digit algorithms (as will be seen). If each redundant-binary digit is expressed in sign and magnitude form, conversion from SM_r to RB_r mode is trivial and involves appending the single sign bit to each of the k magnitude bits. Conversion from RB_r to SM_r is less trivial, however, and involves recognition that the sign of the radix-2^k digit is that of the most significant non-zero binary digit, followed by subtraction of the magnitudes of those digits of opposite sign from the magnitudes of those binary digits of the same sign.

3.4.2 Number format and range for mantissa - In this thesis, the mantissa is assumed to be represented by a one-origin right indexed d-vector of length n. The radix point is assumed to be at the left of the most significant digit with index one. That is,

$$X^1 = . \; x_1^1 \; x_2^1 \; x_3^1 \; . \; . \; . \; x_{n-1}^1 \; x_n^1$$

For a conventional number representation, the values of digit x_i^1 are $\{0,1,\ldots,r-1\}$ and for the signed-digit format, the digit x_i^1 can assume any value in the digit set $\{(\overline{r-1}),(\overline{r-2}),\ldots,\bar{1},0,1,\ldots,(r-2),(r-1)\}$.

When more than one operand is considered, the superscript is employed to identify a specific operand, i.e., X^1 and X^2 for two operands or $X^1,X^2,\ldots,X^j,\ldots,X^\ell$ for ℓ operands. The i-th digit of x^j is uniquely identified as x_i^j .

The algebraic value of the mantissa is given by

$$X^1 = \sum_{i=1}^{n} x_i^1 \; r^{-i}$$

and $-1 < X^1 < 1$.

3.5 Normalization Considerations

For the preparation of operands and the processing of results, it is necessary to restrict the range of values which the mantissa may assume. One usually restricts this range by requiring that all operands be normalized. This is generally done by defining the form of d-vector representation of the restricted range operands. However, in redundant number representations, there exist pseudo-normal forms, because more than one d-vector representation is possible for the same algebraic value. For example, the two numbers X^1 and X'^1

$$X^1 = .00\ldots1$$

$$X'^1 = .1(\overline{r-1})(\overline{r-1})\ldots(\overline{r-1})$$

have the same algebraic value. The representation X'^{1} satisfies the conventional normalization condition $x_1'^{1} \neq 0$ but not the minimum magnitude ($\geq \frac{1}{2}$) requirements for its algebraic value.

3.5.1 Definition and range of normalized numbers - Three alternative definitions of normalized operands were considered.

Definition 1

A number X (of nonzero algebraic value) is considered normalized when its d-vector representation $X^{\dagger} \equiv .x_1x_2...x_n$ - satisfies either of two conditions

a. $|x_1| \geq 2$

b. $|x_1| = 1$ and $x_1 \cdot x_2 \geq 0$

Definition 2

A number X (of nonzero algebraic value) is considered normalized when in its d-vector representation $X \equiv .x_1x_2...x_{\tau-1}\ x_\tau\ ...\ x_n$ either

a. $|x_1| \geq 2$

or

b. $|x_1| = 1$, $x_2 = x_3 = ... = x_{\tau-1} = 0$ and

$$x_1 \cdot x_\tau > 0 \quad , \quad \tau \leq n$$

$$x_1 \cdot x_\tau = 0 \quad , \quad \tau = n$$

where n is the length of the operand.

---

†In these definitions, the superscript on X has been dropped for ease of readability.

Definition 3

A number X (of nonzero algebraic value) is considered normalized when its d-vector representation $X \equiv . x_1 x_2 x_3 \ldots x_j \ldots x_n$ satisfies the conditions

a. $|x_1| > 0$

and b. $x_1 \cdot x_i \geq 0$

such that $2 \leq i < j$ and x_j is the first (counting from left) zero digit in the d-vector of x. For example $X = .11\bar{1}0\bar{1}$ is considered unnormalized per Definition 3.

The range of values for the normalized operands under Definitions 1 and 3 is

$$\frac{r-1}{r^2} + \frac{1}{r^n} \leq |x| \leq 1 - \frac{1}{r^n}$$

and for operands, normalized according to Definition 2, the range is

$$\frac{1}{r} \leq |x| \leq 1 - \frac{1}{r^n}$$

Note that the Definition 2 is equivalent to the conventional definition.

Of the three definitions, the Definition 3 was adopted. The factors affecting the choice between the three definitions are:

i) Complexity of normalization implementation,

ii) Amount of significance loss,

and iii) Logic complexity of quotient selection.

For normalizing numbers according to Definition 2, one needs to examine more digits than for Definition 1. If immediately following $|x_1| = 1$, there is a string of zeros of length v, the normalization procedure must examine at least v+2 digits (to determine the sign of the

first nonzero digit following the string of zeros) in the case of Definition 2, whereas only 2 digits need to be looked at for Definition 1. Since the examination of digits is essentially a serial process, it takes v extra steps for Definition 2.

When the results are normalized according to Definitions 1 and 3 there may be a potential loss of one extra radix-r significant digit, compared to Definition 2. Such a case can occur when a result d-vector is of the form $1.0\ 0\ldots\bar{x}_i\ x_{i+1}\ldots x_n$ and a post-normalization shift becomes necessary.† However, it is expected that such a case would not occur very often because for higher radix arithmetic the frequency of zero digits is low, and also the overflow occurs less often [37].

Finally, because of the redundant number representation, the quotient is calculated based on a truncated version of the partial remainder and the divisor. The number of digits of the truncated operands necessary for quotient calculation depends on the minimum algebraic value of the truncated divisor, say $\hat{D}_{min}$. The lower the value of $\hat{D}_{min}$, the greater are the number of digits of the truncated divisor and partial remainder necessary for the quotient calculation. For higher values of radix r, e.g., $r \geq 16$, the difference in the minimum value of the truncated, normalized divisor for Definitions 1 and 2 is very small and the number of digits required for quotient calculation remains the same. However, for lower radices ($8 \geq r \geq 2$), the number of digits required and thus the logic complexity for quotient calculation is greater for Definitions 1 and 3 than for Definition 2.

---

†In the case of Definition 2, this number would have to be normalized further to the form $.(r-1)(r-1)\ldots(\overline{x_i-1})\ x_{i+1}\ \ldots x_n$ and a post normalization shift would not be necessary.

From the above discussion of factors affecting the choice of definition of normalized numbers for maximally redundant signed-digit operands, it is clear that any of the three choices would be almost equally useful for higher radices ($r \geq 16$). But for $r = 2,4$ where the probability of a string of zero is higher, Definition 1 or 3 would definitely be better for faster normalization, although the logic complexity of quotient calculation would correspondingly be increased. The speed of quotient calculation and thus the speed of the DIVIDE instruction would be decreased. But the frequency of DIVIDE instructions is rather low compared to ADD instructions and so Definition 1 or 3 would overall add to the speed of the arithmetic processing.

For the present research, Definition 3 was chosen because of its compatibility with the Assimilation Recode (AR) microinstruction's digit algorithm. The Assimilation Recode algorithm converts a signed-digit operand into a conventional sign-magnitude operand. This compatibility allows the sharing of logic in the implementation of Normalize Recode (NR) microinstruction and microinstruction AR and thus reduces the control complexity of a PE. Digit algorithms for microinstructions NR and AR are discussed in Sections 3.6.4 and 3.6.5.

Normalization of an operand is achieved by shifting out leading zeros, followed by a 'Normalize Recode' microinstruction, again followed by shifting out leading zeros, if any. This is discussed further in Section 6.2.6.

3.6 Arithmetic Microinstructions and Corresponding Digit Algorithms

In the present research, the design of the Processing Element is restricted to the capability of performing the four basic arithmetic processes of Addition, Subtraction, Multiplication and Division of two operands. Multiplication and Division are implemented as a number of additions or subtractions (of a multiple of multiplicand or divisor) and shifts as in a classical Von-Neumann Arithmetic Unit. Hence the basic arithmetic microinstruction necessary is of the form

$$X^w = X^u + (X_i^q * X^v) \qquad (3.6.1)$$

where X^w, X^u and X^v are d-vectors and x_i^q is a digit. In case of multiplication (division), X^v is the multiplicand (divisor), X^u is the old partial product (partial remainder) and X^w is the new partial product (partial remainder) and x_i^q is the signed multiplier (quotient) digit.

The microinstruction which achieves (3.6.1) is termed 'Form Multiple and Add' (FMA).

Other microinstructions of an arithmetical nature which are needed for the execution of four basic arithmetic processes are Simple Sum (SS), Multiple Sum (MS), Normalize Recode (NR), and Assimilation Recode (AR). The function of each of these microinstructions and the corresponding digit algorithm for execution of the microinstruction in a processing element is discussed next.

3.6.1 Simple sum (SS) microinstruction - This microinstruction forms the sum of two signed-digit operands say A and Φ such that

$$A' = A + \Phi$$

where A´ is the new value of the operand A. In general, A and A´ are in the Accumulator register of the Processing Elements.

At the digit level, the SS microinstruction is characterized by

$$a_i' = a_i + \phi_i + T_i - r\,T_{i-1}$$

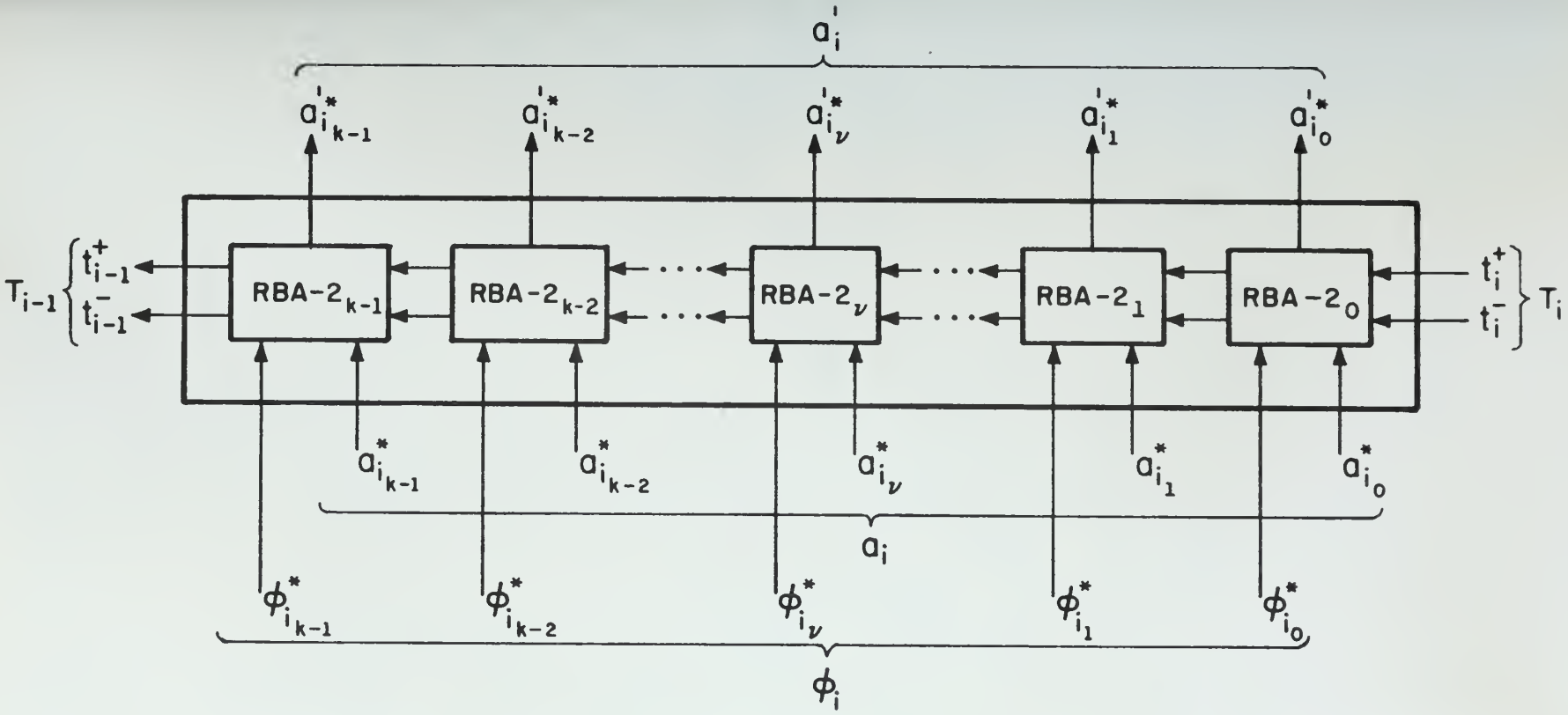
where a_i, a_i' and ϕ_i are radix-r signed digits of the operand in the active registers of PE_i, T_i is the 'Transfer' (carry-borrow) from the adjacent processing element PE_{i+1} and T_{i-1} is the 'Transfer' out of the PE_i.

3.6.1.1 Digit algorithm - The specification of the digit algorithm for SS is intimately connected with its implementation and is described below in terms of its algebraic implementation.

Because of the structural regularity requirements of the LSI technology, the sum of two radix-r signed digits a_i and ϕ_i is realized in a linear cascade of k, two input redundant binary adders.

This is schematically shown in Figure 3.1. RBA-2 is a two input redundant binary adder which accepts two redundant binary digits $a^*_{i_\nu}$, $\phi^*_{i_\nu}$ $\varepsilon \{\bar{1},0,1\}$ and produces one redundant binary digit. The design of such an RBA-2 was studied in detail by Borovec [38] and we shall interchangeably use the term Borovec Unit (BU) for RBA-2.

3.6.1.1.1 Arithmetic design of RBA-2 - The major consideration in the design of RBA-2 was the minimization of the number of pins required for the 'Transfer' into and out of an RBA-2. One such design is shown in Figure 3.2. RBA-2 is realized by a series of four arithmetic transformations as follows.



$$a_i = \sum_{\nu=0}^{k-1} a_{i_\nu}^* \cdot 2^\nu \qquad T_i,\ T_{i-1} \in \{\bar{1},0,1\}$$

$$\phi_i = \sum_{\nu=0}^{k-1} \phi_{i_\nu}^* \cdot 2^\nu \qquad t_i^+,\ t_{i-1}^+ \in \{0,1\}$$

$$a_{i_\nu}^*,\ \phi_{i_\nu}^* \in \{\bar{1},0,1\} \qquad t_i^-,\ t_{i-1}^- \in \{0,\bar{1}\}$$

Figure 3.1 Illustration of Digit Algorithm for Micro-instruction SS.

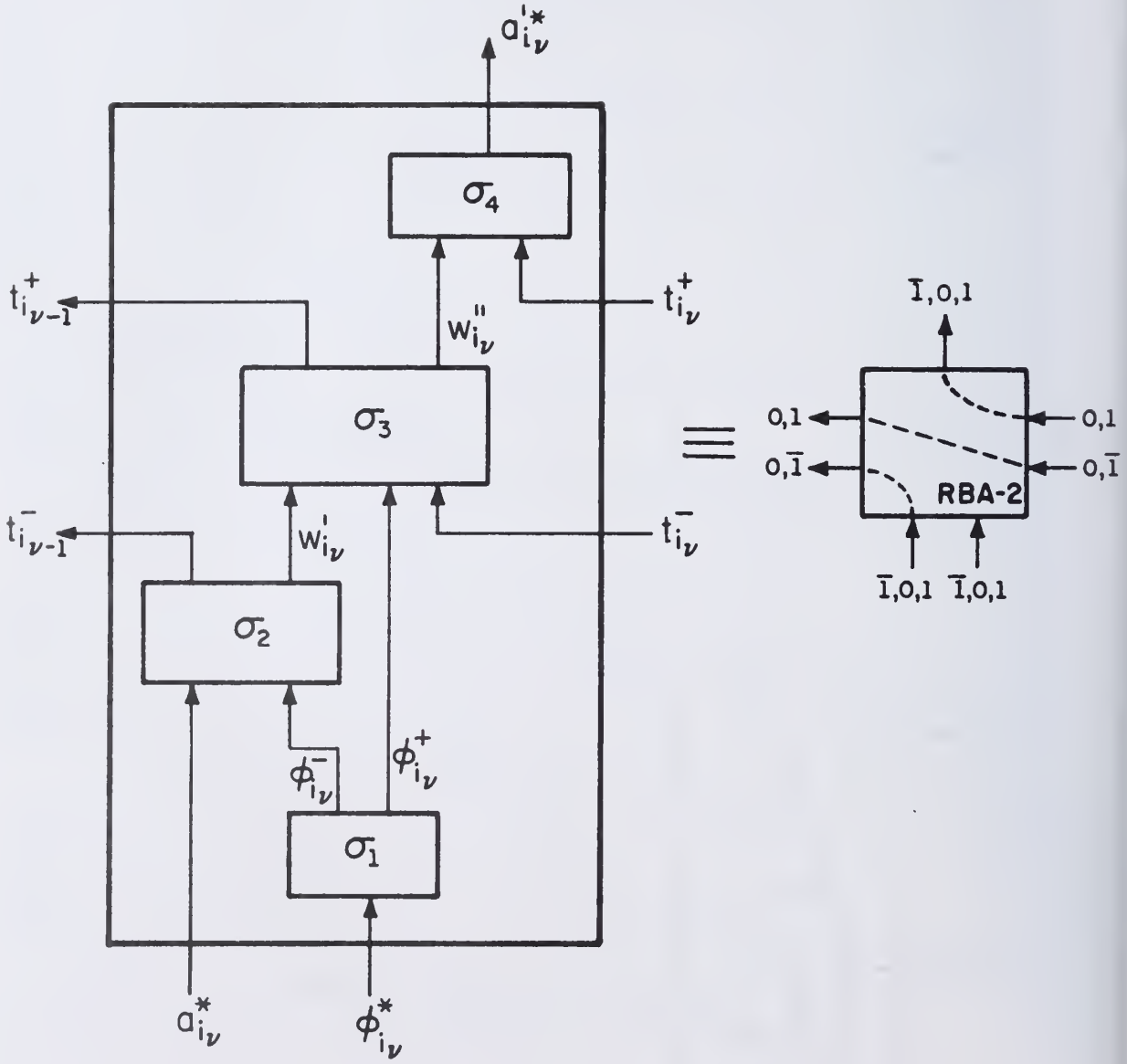


Figure 3.2 Arithmetic Structure of an RBA-2.

$$\sigma_1 : \phi^*_{i_\nu} = \phi^+_{i_\nu} + \phi^-_{i_\nu}$$

$$\sigma_2 : a^*_{i_\nu} + \phi^-_{i_\nu} = w'_{i_\nu} + 2\, t^-_{i_\nu}$$

$$\sigma_3 : w'_{i_\nu} + \phi^+_{i_\nu} + t^-_{i_\nu} = w''_{i_\nu} + 2t^+_{i_{\nu-1}}$$

$$\sigma_4 : w''_{i_\nu} + t^+_{i_\nu} = a'^*_{i_\nu}$$

where $a^*_{i_\nu}$, $\phi^*_{i_\nu}$ and $a'^*_{i_\nu} \in \{\bar{1},0,1\}$

$t^-_{i_\nu}$, $t^-_{i_{\nu-1}}$, $\phi^-_{i_\nu} \in \{0,\bar{1}\}$

and $t^+_{i_\nu}$, $t^+_{i_{\nu-1}}$, $\phi^+_{i_\nu} \in \{0,1\}$

Let us call $t^-_{i_\nu}$, $t^-_{i_{\nu-1}}$ 'negative transfers' and $t^+_{i_\nu}$, $t^+_{i_{\nu-1}}$ 'positive transfers'. The logic design of RBA-2 is discussed in Section 4.2.2.3. It is clear from the design of RBA-2 above that the transfer digits $t^-_{i_\nu}$ and $t^+_{i_\nu}$ are respectively dependent on the inputs to the RBA-2s which are immediately adjacent and one next to it.

In terms of the notation of Section 2.4, we have for the SS micro-instruction

$$SS^F_i = SS^F_{i-1} \qquad \text{for all } i,\ 1 \leq i \leq n$$

$$SS^F_0 = \langle \text{null} \rangle .$$

$$SS^G_{i+1} = \{t^+_i, t^-_i\}$$

and $\alpha_{SS} = 2$ for $r = 2$

$= 1$ for $r > 2$

3.6.2 Form multiple and add (FMA) microinstruction - This microinstruction is used to form the product of the multiplicand (divisor) d-vector and a multiplier (quotient) digit, which when added to the old partial product (partial remainder) gives the new partial product (partial remainder) in the execution of a Multiplication (Division) of two d-vector operands.

At the digit level, this microinstruction is characterized by the arithmetic transfer function

$$a_i' = a_i + m_j \cdot \phi_i - r\, T_{i-1} + T_i \qquad (3.6.3)$$

where

a_i, ϕ_i are the digits in the active operand registers of the processing element PE_i

a_i' is the new value of digit a_i

m_j is a multiplier (quotient) digit

r is the radix

and T_i(T_{i-1}) is the 'Transfer' (carry/borrow) from (to) adjacent processing element PE_{i+1} (PE_{i-1}).

This is functionally represented in Figure 3.3.

3.6.2.1 Digit algorithm - The major considerations in the design of the digit algorithm for microinstruction FMA were the LSI technology constraints--namely, that the implementation logic for FMA should consist of a regular and repetitive structure. The specification of the digit algorithm is intimately connected with the implementation procedure and is described below as such.

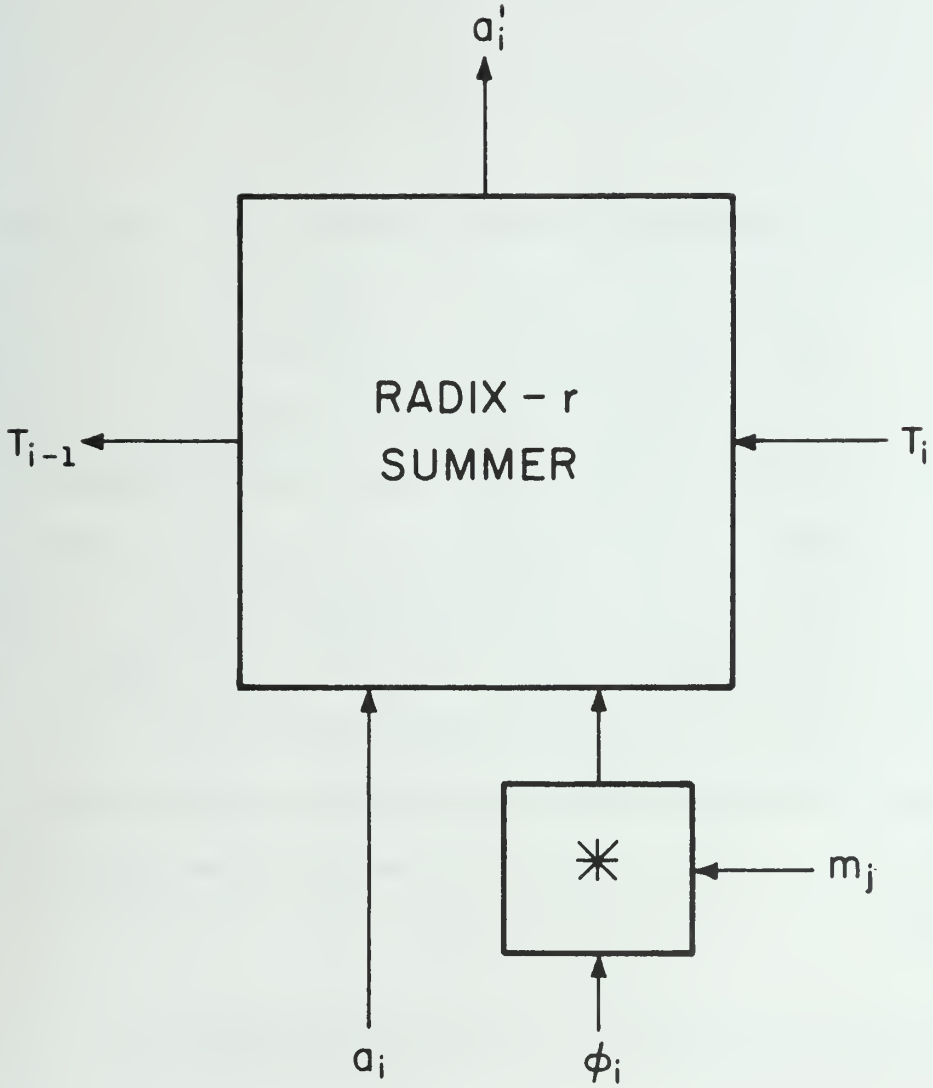


Figure 3.3 Functional Representation of Micro-instruction FMA.

The transfer function in Equation (3.6.3) is achieved by a series of two transformations f_1 and f_2 as shown in Figure 3.4. The two transformations are

$$f_1 : m_j \cdot \phi_i = r\, t^P_{i-1} + w_i \qquad (3.6.4)$$

and

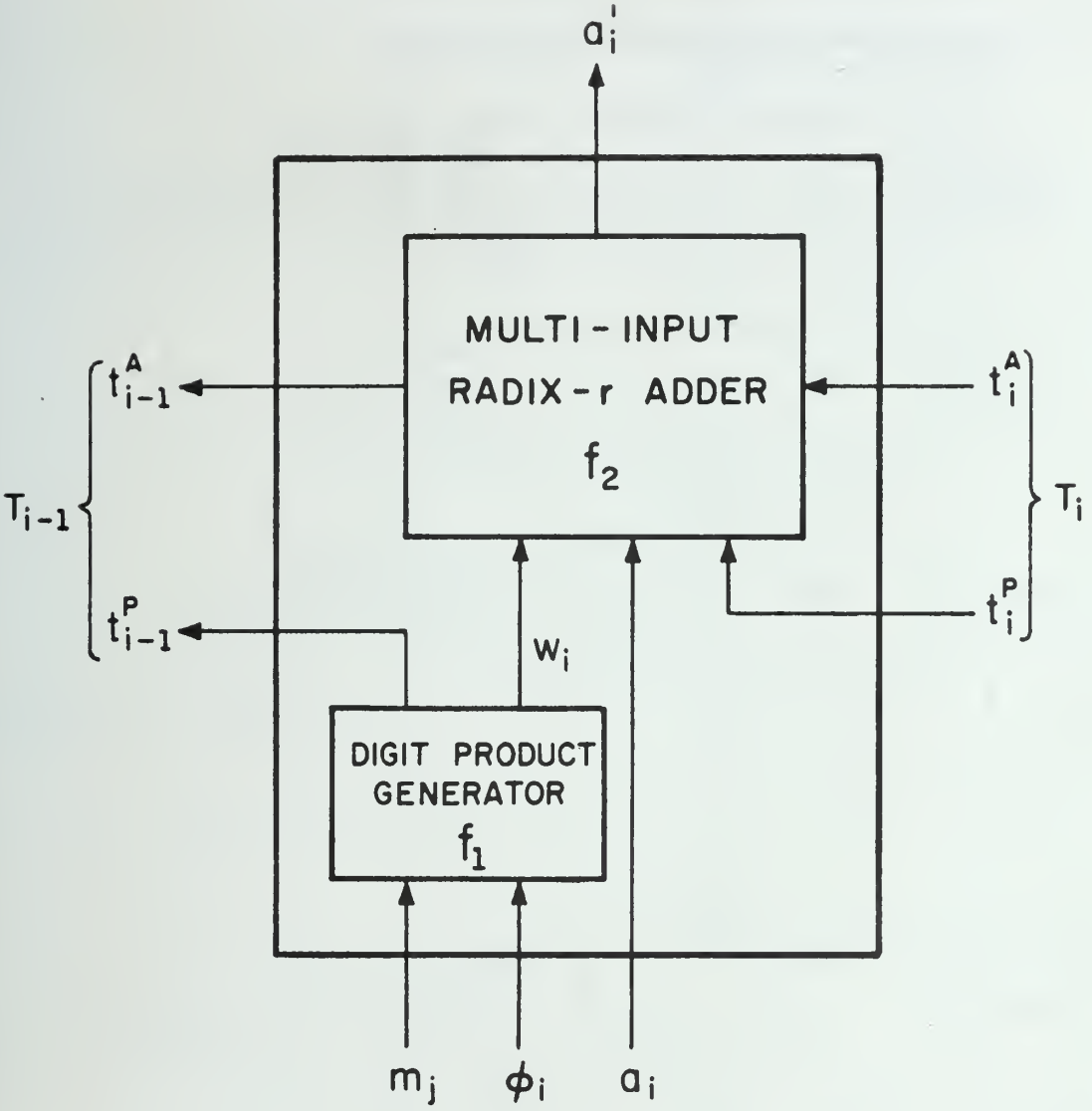
$$f_2 : w_i + a_i + t^P_i + t^A_i = a'_i + r\, t^A_{i-1} . \qquad (3.6.5)$$

Transformation f_2 essentially requires a radix-r multi-input adder which forms the sum of digits of both signs. This multi-input adder is implemented as a k-stage linear cascade of radix-2 multi-input adder where each input of a radix-2 adder can assume three values $\bar{1},0,1$. The input digits w_i, a_i, t^P_i are expressed in the form of radix-2 d-vectors such that each component of the radix-2 d-vector is from the redundant binary digit set $\{\bar{1},0,1\}$. This is schematically shown in Figure 3.5. MIRBA represents the Multi-Input Redundant Binary Adder. The number of redundant binary inputs to each MIRBA are determined as follows. Two algorithms were studied for the implementation of transformation f_1 given in Equation (3.6.4). They differ in the maximum values that w_i and t^P_{i-1} can assume.

3.6.2.1.1 Algorithm 1 - To illustrate the principle,

Let $$\phi_i = \sum_{\ell=0}^{k-1} \phi^*_{i_\ell} \cdot 2^\ell$$

$$m_j = \sum_{q=0}^{k-1} m^*_{j_q} \cdot 2^q \qquad \phi^*_{i_\ell},\ m^*_{j_q} \in \{\bar{1},0,1\}.$$



$$f_1: \quad m_j \cdot \phi_i = r\, t_{i-1}^P + w_i$$

$$f_2: \quad w_i + a_i + t_i^P + t_i^A = a_i' + r\, t_{i-1}^A$$

Figure 3.4 Functional Representation of the Digit Algorithm for FMA.

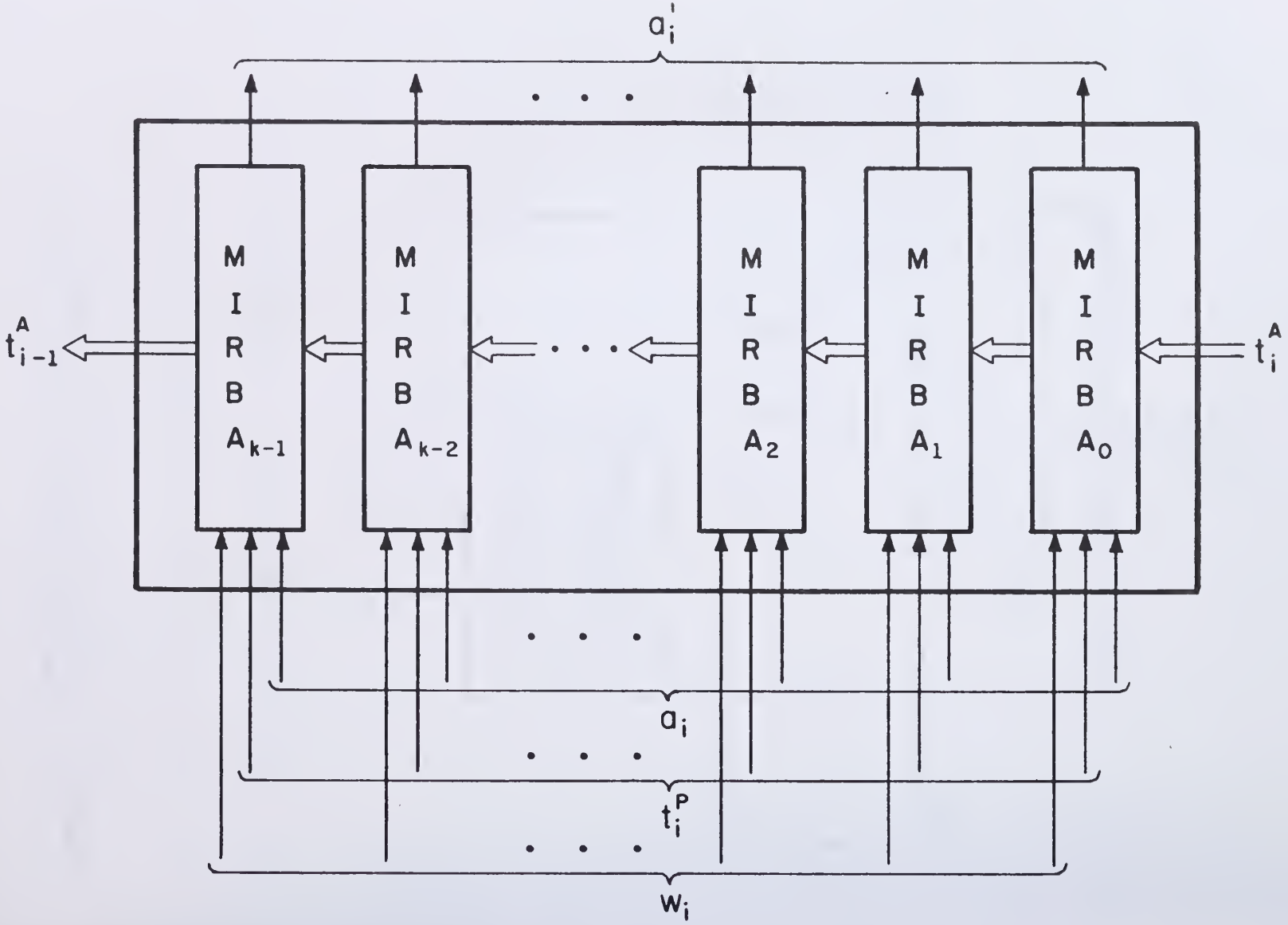


Figure 3.5 Functional Representation of Transformation f_2.

The product $\phi_i \cdot m_j$ is implemented by a product matrix generator which consists of a k x k square array of redundant binary product cells. Each cell performs the product of two redundant binary digits $\phi^*_{i_\ell}$ and $m^*_{j_q}$ and its output product digit $p^*_{\ell q}$ is also in the digit set $\{\bar{1},0,1\}$.

The product may be viewed in terms of the sums of the $p^*_{\ell q}$ terms of the same weight in the product matrix.

$$\phi_i \cdot m_j = \sum_{\nu=0}^{2k-2} \left(\sum_{\ell=0}^{\nu} \phi^*_{i_\ell} \cdot m^*_{j_{\nu-\ell}} \right) 2^\nu$$

$$= \sum_{\nu=0}^{2k-2} 2^\nu \left(\sum_{\ell=0}^{\nu} p^*_{\ell,\nu-\ell} \right) \qquad (3.6.6)$$

where $p^*_{\ell,\nu-\ell} = \phi^*_{i_\ell} \cdot m^*_{j_{\nu-\ell}}$

and $p^*_{\ell,\nu-\ell}$ does not exist when either $\ell > k-1$

or $\nu-\ell > k-1$.

The number N_ν of product elements in the ν-th column of the product matrix is given by

$$N_\nu = \begin{cases} \nu+1 & 0 \leq \nu \leq k-1 \\ -\nu + (2k-1) & k \leq \nu \leq 2k-2 \end{cases} \qquad (3.6.7)$$

The number N_ν is maximum in column of weight 2^{k-1} and is equal to k. The product elements in other columns decrease uniformly by one on either side of this column as shown in Figure 3.6.

$$
\begin{array}{llllll}
 & \phi^*_{i_{k-1}} & \phi^*_{i_{k-2}} & \cdots & \phi^*_{i_1} & \phi^*_{i_0} \\
 & m^*_{j_{k-1}} & m^*_{j_{k-2}} & \cdots & m^*_{j_1} & m^*_{j_0} \\
\hline
 & p^*_{k-1,0} & p^*_{k-2,0} & \cdots & p^*_{10} & p^*_{00} \\
p^*_{k-1,1} & p^*_{k-2,1} & p^*_{k-3,1} & \cdots & p^*_{01} & \\
\vdots & \vdots & & & & \\
p^*_{k-1,k-1} & p^*_{k-2,k-1} & \cdots & p^*_{1,k-1} & p^*_{0,k-1} & \\
\hline
 & p^*_{\ell q} = \phi^*_{i_\ell} \cdot m^*_{j_q} & & & &
\end{array}
\qquad \phi^*_{i_\ell},\ m^*_{j_q},\ p^*_{\ell_q} \in \{\bar{1}, 0, 1\}
$$

Figure 3.6 A Redundant Binary Product Matrix.

Equation (3.6.6) can be rewritten in the following form

$$\phi_i \cdot m_j = \sum_{\nu=0}^{k-1} 2^{\nu} \left(\sum_{\ell=0}^{\nu} p^*_{\ell,\nu-\ell} \right) + \sum_{\nu=k}^{2k-2} 2^{\nu} \left(\sum_{\ell=0}^{\nu} p^*_{\ell,\nu-\ell} \right)$$

$$= \sum_{\nu=0}^{k-1} 2^{\nu} \left(\sum_{\ell=0}^{\nu} p^*_{\ell,\nu-\ell} \right) + 2^{k} \sum_{\nu=k}^{2k-2} 2^{\nu-k} \left(\sum_{\ell=0}^{\nu} p^*_{\ell,\nu-\ell} \right) \tag{3.6.8}$$

The columns of weight 2^{ν} ($k \leq \nu \leq 2k-2$) of the product matrix can be considered as forming a carry t^P_{i-1} called Collective Product Transfer, CPT to the next more significant radix-2^k digital position i-1. These ($k \leq \nu \leq 2k-2$) CPT columns have weights $2^{\nu-k}$ (Equation (3.6.8)) with respect to the higher significant digital position. When similar CPT columns from digital position i+1 are added in the appropriate (of the same weight) MIRBA of the digital position i, all the stages of the linear cascade of MIRBAs in PE_i become identical, each MIRBA having k-inputs. This is illustrated in Figure 3.7.

Further, the transformation f_2 requires the addition of one radix-r digit a_i to w_i and t^P_i, and the digit a_i contributes one redundant binary input to each position of MIRBA. Hence the transformation f_2 requires k MIRBAs, each capable of summing k+1 redundant binary inputs, as well as the 'Transfer' from the adjacent MIRBA position. Figure 3.8 schematically shows the implementation of FMA digit algorithm for radix 16, that is, k=4.

Values of $|w_i|_{max}$ and $|t^P_{i-1}|_{max}$

From Equations (3.6.4) and (3.6.8), we have

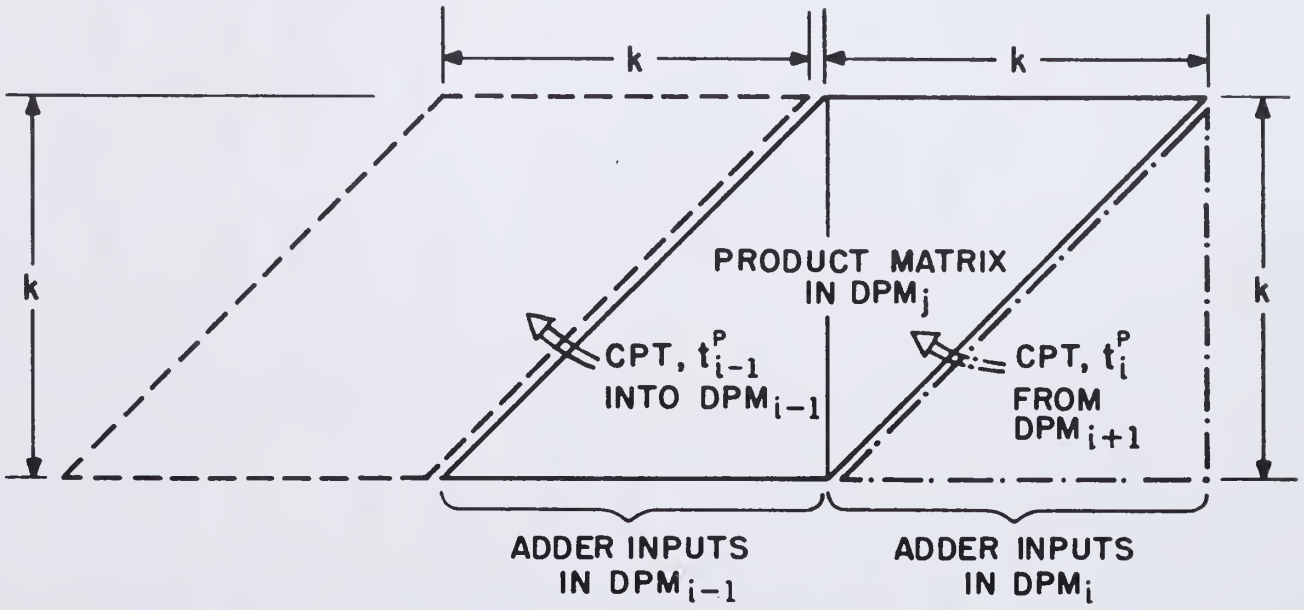
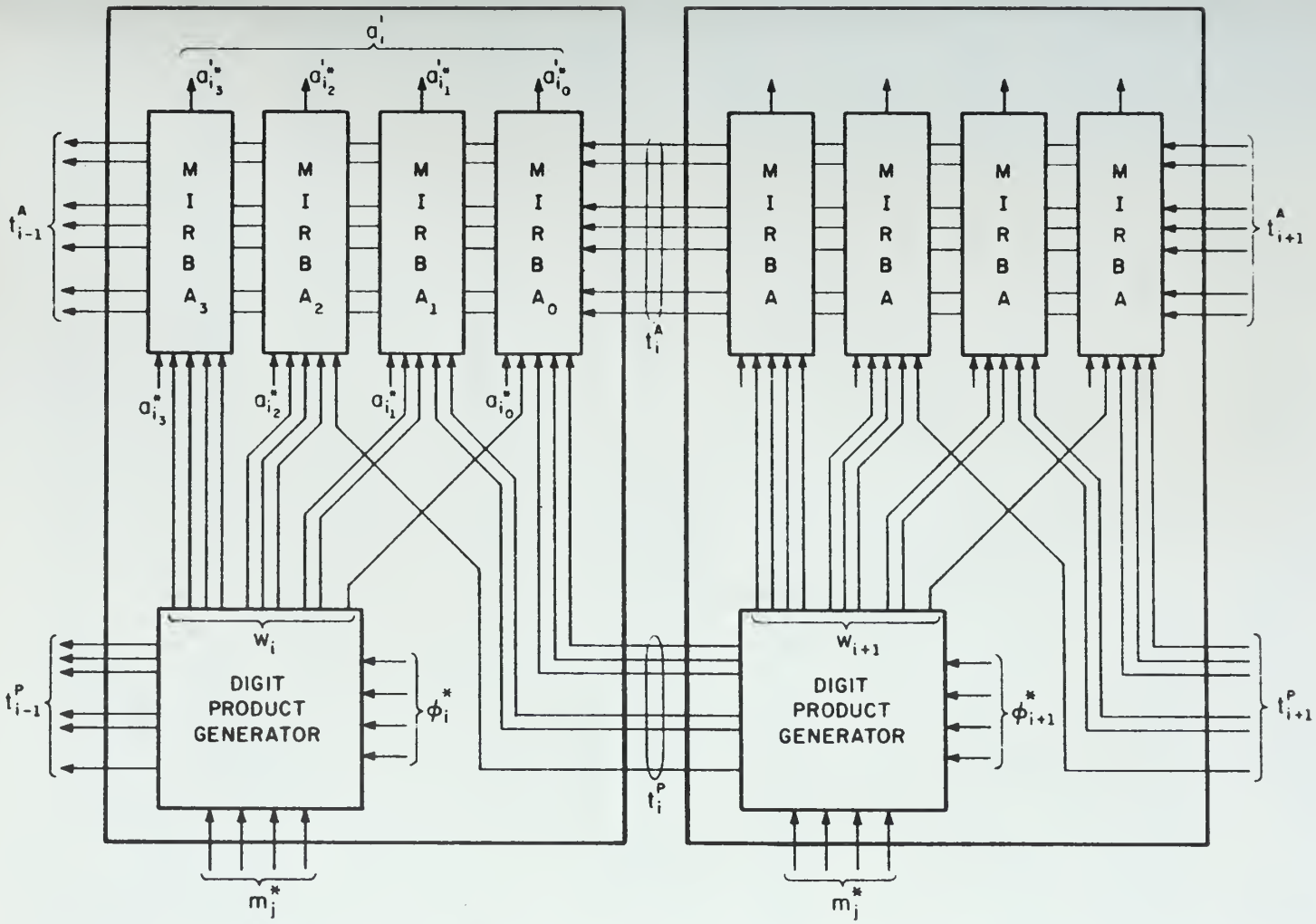


Figure 3.7 Illustration of Adjacent Overlapping Product Matrices and 'Collective Product Transfer, CPT'.



Note: t_i^P, m_j^* and ϕ_i^* are in Redundant Binary Format, RB_r.

Figure 3.8 Illustration of the Implementation of Algorithm 1 of Microinstruction FMA, using Redundant Binary Product Matrix Generator. (Radix = 16)

$$w_i = \sum_{\nu=0}^{k-1} 2^{\nu} \left(\sum_{\ell=0}^{\nu} p^*_{\ell,\nu-\ell} \right) = \sum_{\nu=0}^{k-1} 2^{\nu} \left(\sum_{\ell=0}^{\nu} \phi^*_{i_\ell} \cdot m^*_{j_{\nu-\ell}} \right) \tag{3.6.9}$$

$$t^P_{i-1} = \sum_{\nu=k}^{2k-2} 2^{\nu-k} \left(\sum_{\ell=0}^{\nu} p^*_{\ell,\nu-\ell} \right) \sum_{\nu=k}^{2k-2} 2^{\nu-k} \left(\sum_{\ell=0}^{\nu} \phi^*_{i_\ell} \cdot m^*_{j_{\nu-\ell}} \right) \tag{3.6.10}$$

From Equations (3.6.9) and (3.6.7), we have

$$|w_i|_{max} = 1 \cdot 2^0 + 2.2^1 + \ldots + (.\nu+1)\ 2^{\nu} + \ldots + (k).2^{k-1}$$

$$= 2^k(k-1) + 1 \tag{3.6.11}$$

Further

$$|w_i|_{max} + 2^k\ |t_{i-1}|_{max} = (2^k-1)^2$$

$$\therefore\ |t^P_{i-1}|_{max} = \frac{(2^k-1)^2 - (2^k(k-1)+1)}{2^k}$$

$$= 2^k - (k+1) \tag{3.6.12}$$

In summary, the digit algorithm 1 for the microinstruction FMA can be described as follows:

i) Perform transformation f_1 by recoding the product of digits ϕ_i and m_j into w_i, t^P_{i-1} such that w_i and t^P_{i-1} are given by Equations (3.6.9) and (3.6.10) respectively.

ii) Perform transformation f_2 in a k-stage linear cascade of (k+1) input redundant binary adder.

The design of the multi-input redundant binary adder is discussed in Section 3.6.2.1.3.

3.6.2.1.2 Algorithm 2 - In this algorithm, the transformation f_1 recodes the product $\phi_i \cdot m_j$ into digits w_i and t^P_{i-1} such that

$$w_i \quad \varepsilon \{(\overline{r-1}),(\overline{r-2}),....,\bar{1},0,1,....,(r-2),(r-1)\}$$

and $$t^P_{i-1} \varepsilon \{(\overline{r-2}),....,\bar{1},0,1,....,(\overline{r-2})\}.$$

Clearly, the recoded digits w_i, t^P_{i-1} contribute only one redundant binary input to each MIRBA of the linear cascade.

Then the transformation f_2 is performed in the k-stage linear cascade of 3 input MIRBAs. This is illustrated in Figure 3.9.

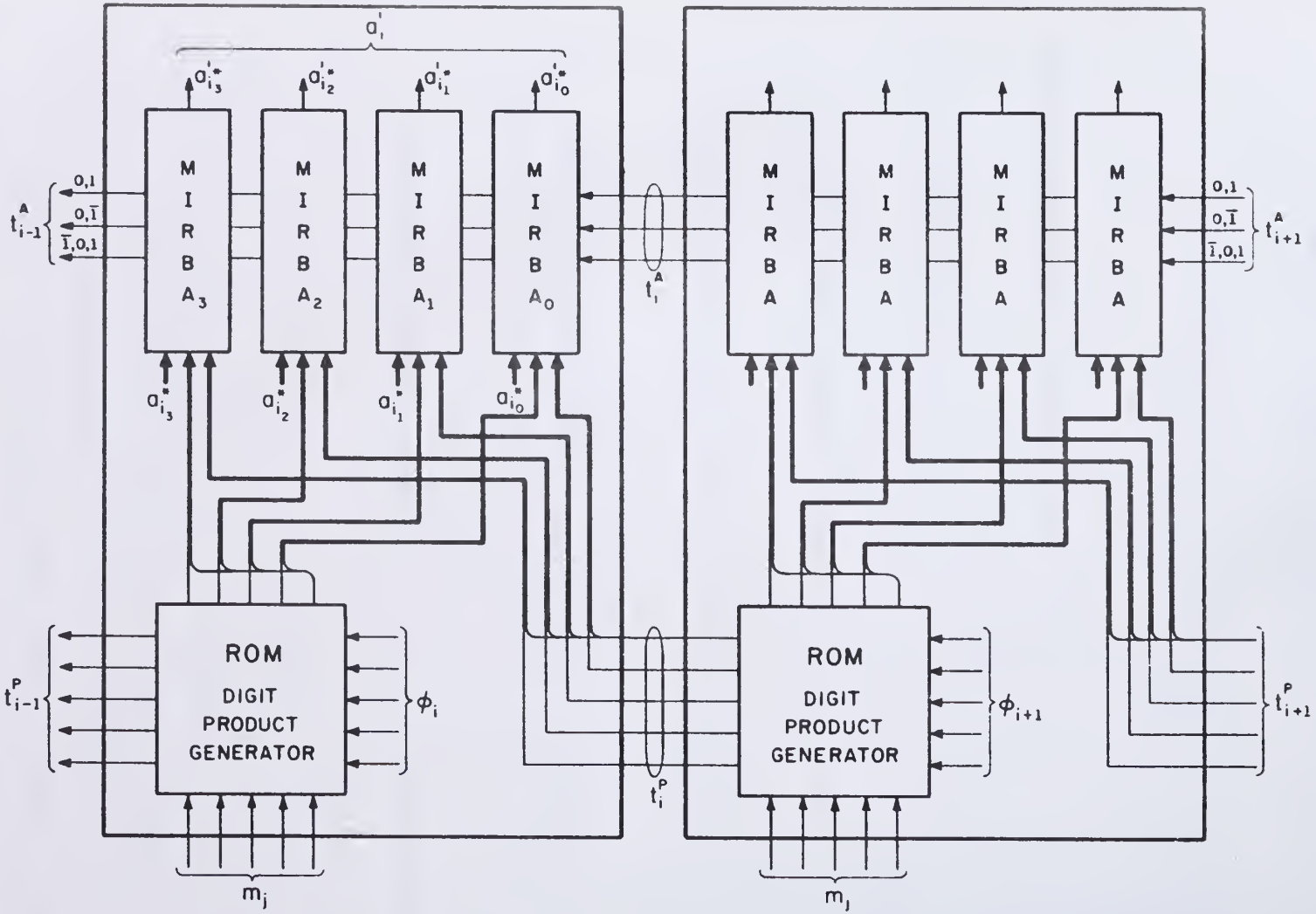
Note that, in algorithm 2 the number of inputs to the MIRBAs is always three, independent of the value of k.

The LSI implications of algorithms 1 and 2 are discussed later in Section 4.2.2.5.

3.6.2.1.3 Design of a multi-input redundant binary adder (MIRBA) - A MIRBA is a limited carry/borrow propagation adder which accepts several redundant binary inputs (digit set $\{\bar{1},0,1\}$) and produces one redundant binary output (with appropriate adder 'Transfers' for more significant adjacent adder stages).

Definition

Let us define a new parameter α^b. The redundant binary output of any MIRBA is dependent on the 'Transfers' (the composite term for carry/borrow) input to that MIRBA. In a redundant number system, the 'Transfers' are functions of 'primary' inputs (other than 'Transfer' inputs) to only



Note: Product Transfer t_i^P is in Sign-Magnitude Format, SM_r.

Figure 3.9 Illustration of the Implementation of Algorithm 2 of Microinstruction FMA using ROMs. (Radix = 16)

a limited number of adjacent less significant MIRBAs. α^b denotes the number of such adjacent MIRBAs whose 'primary' inputs in cooperation with the primary inputs of a given MIRBA determine the output of that MIRBA.

The radix-2^k digit processing logic in, say PE_i consists of a k stage linear cascade of (k+1) input MIRBAs. Except for the most significant MIRBA in k-stage cascade, the primary inputs to the MIRBAs in PE_i are functions of radix-2^k operand digits in PE_i and PE_{i+1} (accumulator digits a_i, multiplier digit m_j and multiplicand digits ϕ_i, ϕ_{i+1}). Thus α_j is related to α^b by Equation (3.6.13)

$$\alpha_j = \left\lceil \frac{\alpha^b - 1}{k} \right\rceil + 1 \qquad (3.6.13)$$

3.6.2.1.3.1 Rohatsch's [39] technique - This is a deterministic and explicit transformation procedure which converts a given input digit set into the required output digit set by a series of simple transformations.

In using this technique, one generally proceeds backwards; namely, consider the transformations going from output set to input set. The basic concept of Rohatsch's technique is very simple:

i) Take the desired output set S, find two or more sets A_0, A_1, $A_2,\ldots,A_n$ such that

$$S = A_n + A_{n-1} + \ldots + A_2 + A_1 + A_0.$$

ii) Form the input set M where

$$M = r^n A_n + r^{n-1} A_{n-1} + \ldots + A_1 r^1 + A_0$$

where r is the radix of the adder. In our case, for MIRBA, r=2.

iii) If necessary, repeat the steps i) and ii) (using the last input set as the new output set) as many times as is required to generate a set which includes the desired input set.

Steps i) and ii) above together constitute an n-th order Simple Transformation (referred to as S.T.). For the contiguity of sets M and S, A_n, A_{n-1}, ...,A_0 must be contiguous and the number of distinct digits in sets A_i, $n-1 \geq i \geq 0$ should be greater than or equal to r.

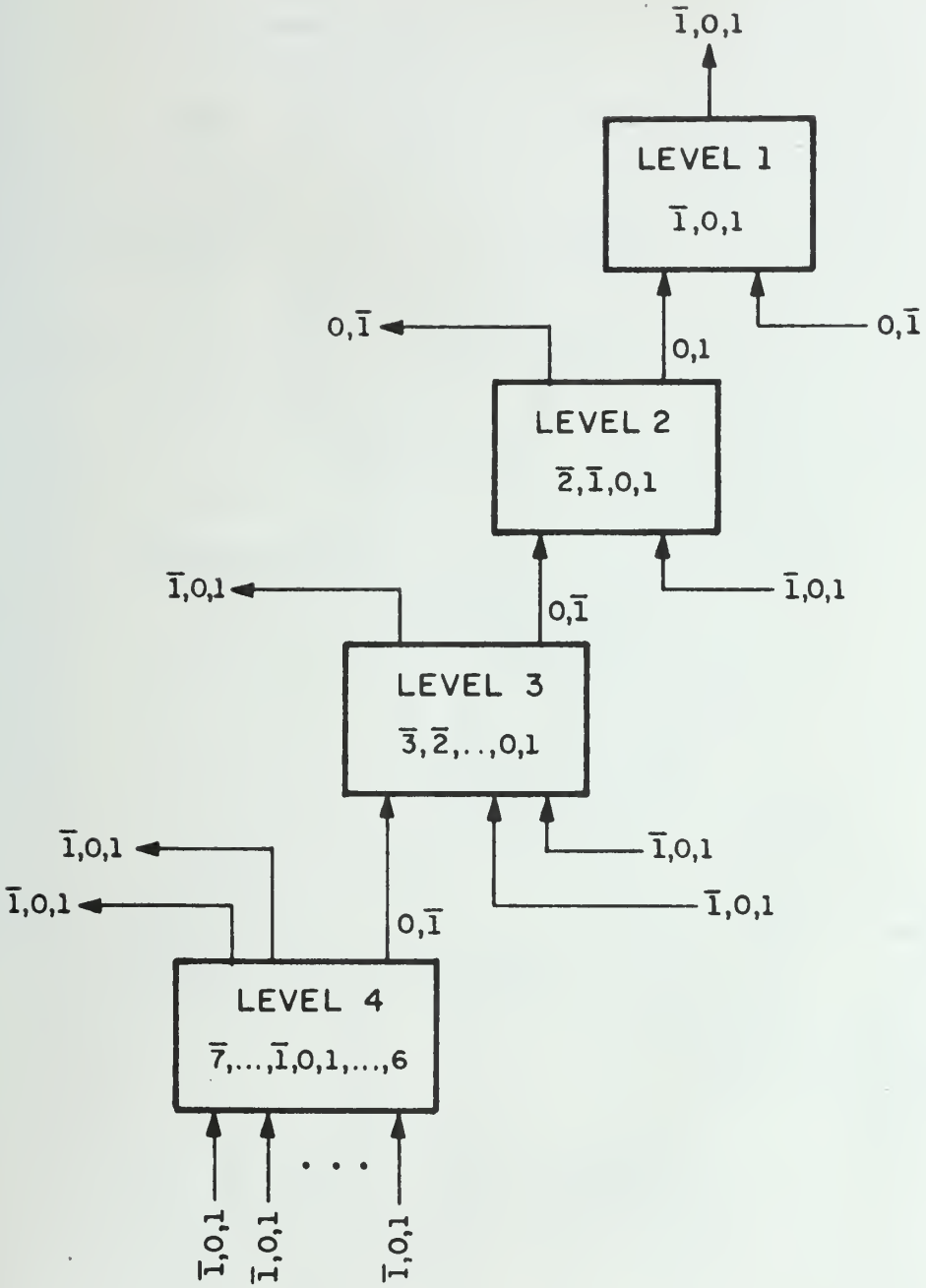
Using the above approach, we find that for $k > 2$, a (k+1)-input MIRBA requires a series of three S.T.s. Figure 3.10a shows one such four level (each level indicated by a box) adder which is applicable for $k \leq 5$. In this, level 1 and level 2 perform first order S.T.s whereas level 3 represents a 2nd order transformation. If level 3 performs a third order or fourth order transformation, such a four level adder would be applicable for $k \leq 9$ and $k \leq 11$ respectively.

It is interesting to note that if level 2 achieves a 2nd order S.T. and level 3 constitutes a 6th order S.T., then the four level adder can be used to sum as much as 51 redundant binary $\{\bar{1},0,1\}$ inputs. This is shown in Figure 3.10b.

However, the logic design of the bottom two levels is highly complicated for $k \geq 5$ if they are to be implemented in two or three logic levels. In practice, the technique is to break down the bottom level structure into equivalent simpler structures frequently at the cost of increasing the number of levels, as shown in Figure 3.10c for k = 5.

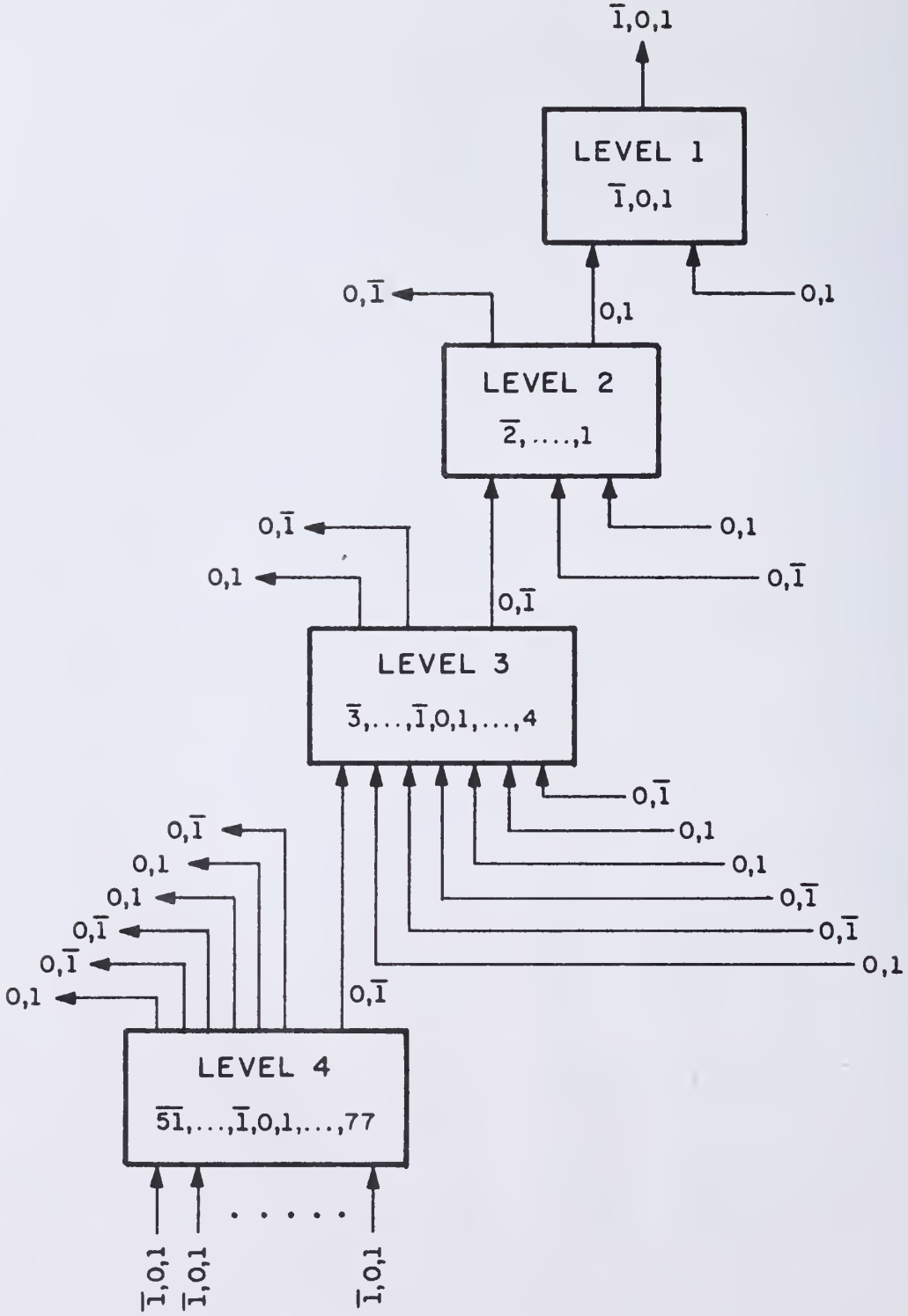
In this adder structure, α^b is given by

$$\alpha^b = \sum_{v=1}^{q-1} n_v$$



Note: Entries in the box show the allowed output digit set values.

Figure 3.10a Illustration of the Algebraic Design of a MIRBA, using First Order Simple Transformations only.



Note: Entries in the box show the allowed output digit set values.

Figure 3.10b Illustration of the Algebraic Design of a MIRBA using Higher (≥ 2) Order Simple Transformation.

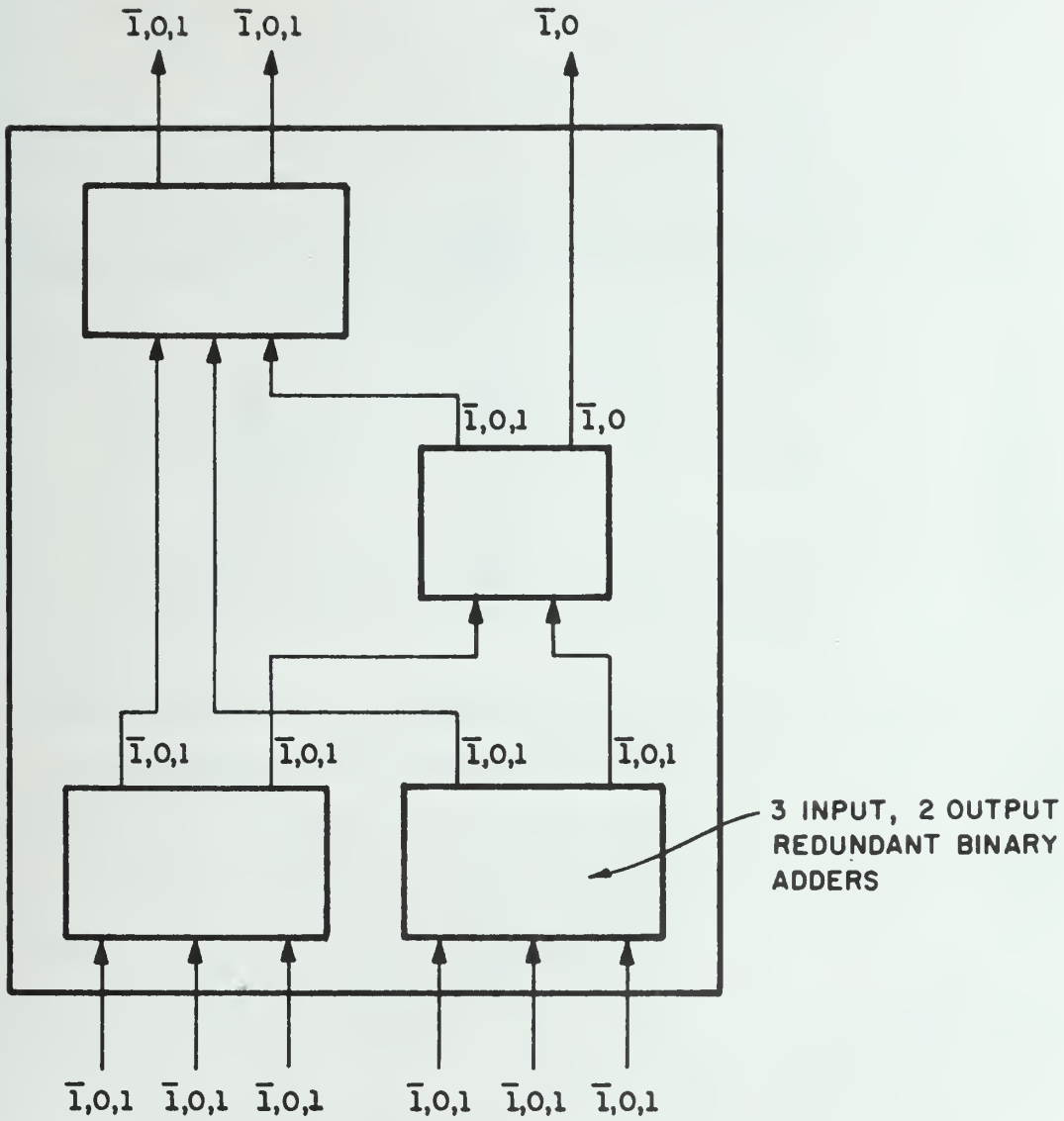


Figure 3.10c Algebraic Design of Bottom Level (Level 4) Box of Figure 3.10a.

where q = number of levels in MIRBA

n_v = order of S.T. performed by adder level v.

Table 3.1 shows the values of α^b and α_j for various values of k, for a (k+1)-input MIRBA.

Table 3.1

Values of α^b and α_j for Various (k+1)-Input MIRBA Configurations

radix	$r = 2^k$	Rohatsch's Technique		log-sum tree		RBA-3,RBA-2 tree structure	
r	k	α^b	α_j	α^b	α_j	α^b	α_j
4	2	3	2	4	3	3	2
8	3	4	2	4	2	5	3
16	4	4	2	6	3	5	2
32	5	4	2	6	2	5	2
64	6	5	2	6	2	6	2
128	7	5	2	6	2	6	2
256	8	5	2	8	2	6	2

3.6.2.1.3.2 Log-sum tree technique - A conceptually simple approach is to realize the (k+1) input MIRBA by a log-sum tree structure of two input redundant binary adders (RBA-2). For a (k+1) input MIRBA, the tree structure has t levels of Borovec Units such that

$$t = \lceil \log_2(k+1) \rceil$$

and the number of BUs required is k. Figure 3.11 shows the log-sum tree structure for a five input MIRBA.

In this configuration,

$$\alpha^b = 2t = 2\lceil \log_2(k+1) \rceil \quad \text{and}$$

$$\alpha_j = \left\lceil \frac{2\lceil \log_2(k+1) \rceil - 1}{k} \right\rceil + 1.$$

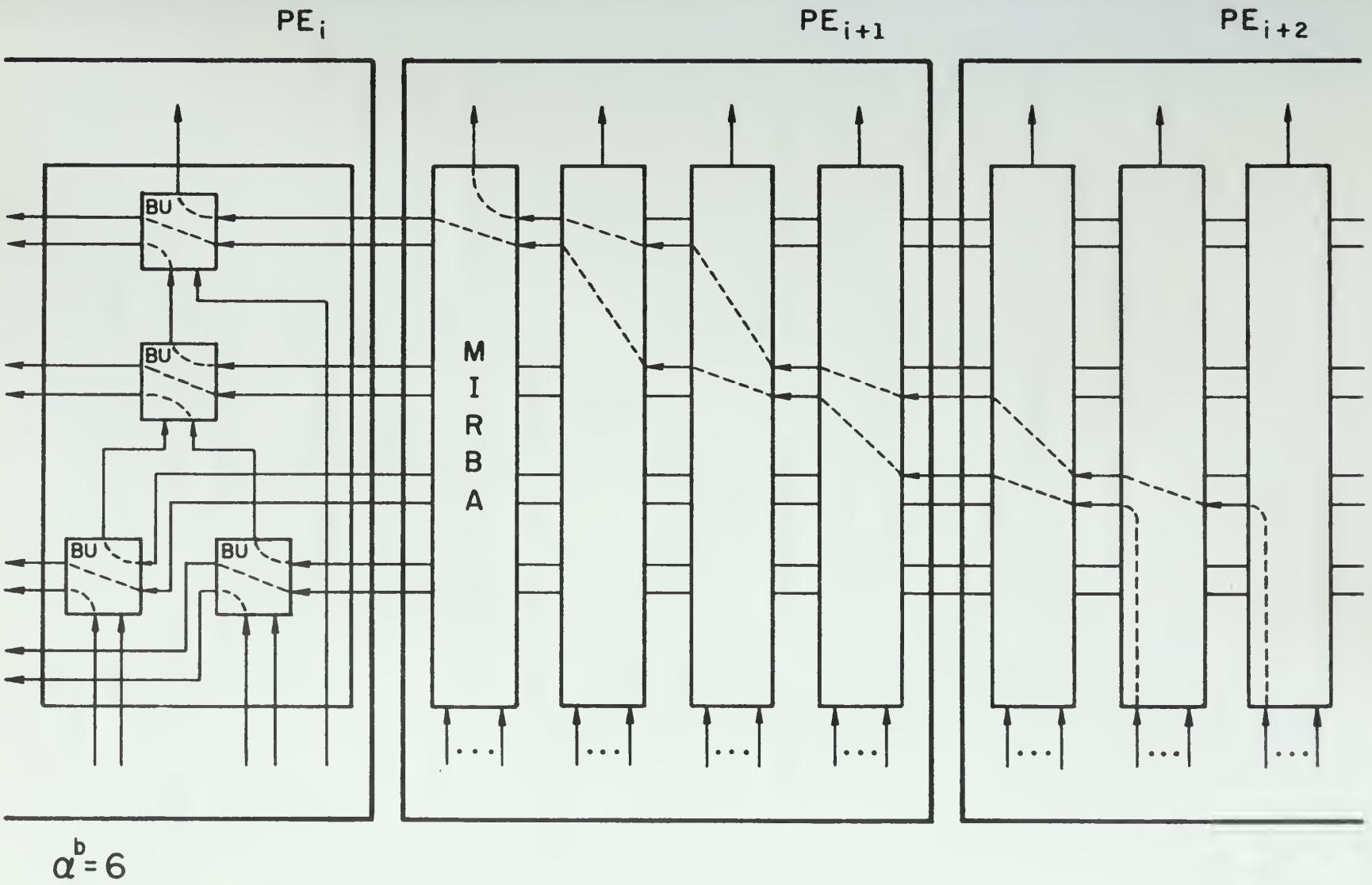


Figure 3.11 Illustration of Log-Sum Tree Structure for a MIRBA using RBA-2s only. (k = 4)

The value of α_j for various values of k is tabulated in Table 1. From the table we find that for k=2 and k=4, that is, radices 4 and 16, the value of $\alpha_j = 3$ and $\alpha_j = 2$ for all other values of k. Since minimum value of α_j is desirable, a different arrangement of BUs as described in third approach given next can be used to achieve $\alpha_j = 2$.

3.6.2.1.3.3 Tree-structure using RBA-3s and RBA-2s - In this configuration, 3-input redundant binary adders (RBA-3) and RBA-2s are connected in a tree structure.

An RBA-3 consists of two BUs, a D-element and a C-element arranged as shown in Figure 3.12. The C-element composes two binary inputs $\{0,1;\ 0,\bar{1}\}$ into one redundant binary $\{\bar{1},0,1\}$ output. The lower BU in combination with the C-element and the D-element acts as a redundant binary (3,2) counter. The upper BU forms the sum of the sum-outputs of the lower BUs and the 'Transfer' output of the lower BU of adjacent less significant RBA-3.

For a design of a (k+1) input MIRBA, RBA-3s are used whenever they can be fully utilized, that is, three inputs are available for addition; and RBA-2s are used when only 2-inputs are to be added at any level of the tree structure. (An exception occurs for k=3 where the log-sum tree technique is necessary.) Figure 3.13 shows a 5-input MIRBA using RBA-3s and RBA-2s as building blocks.

The number of BUs required in this technique is also k for a (k+1)-input MIRBA. The number of BU levels is also $2\lceil\log_2(k+1)\rceil$. Table 3.1 shows the values of α^b and α_j for various of k. It shows that $\alpha_j = 2$ for all values of k except k=3.

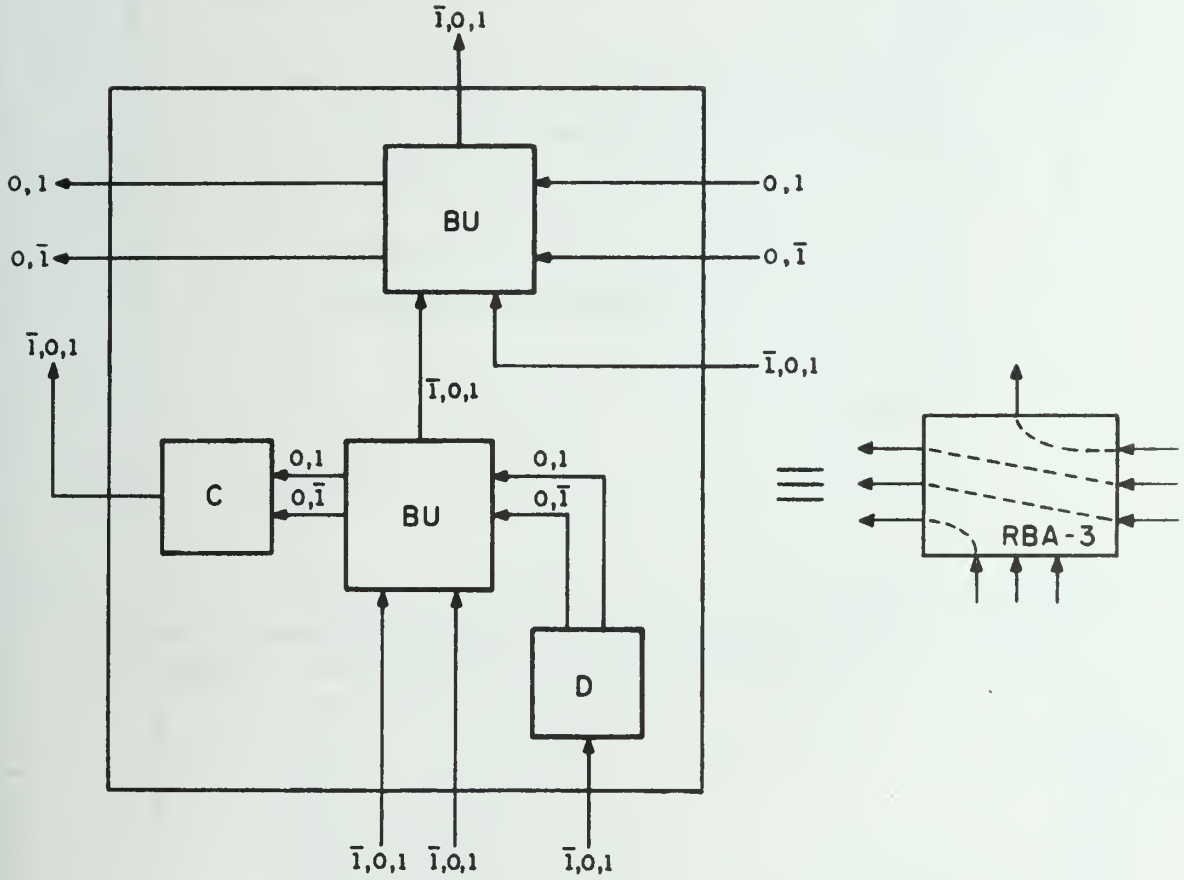


Figure 3.12 Arithmetic Structure of an RBA-3.

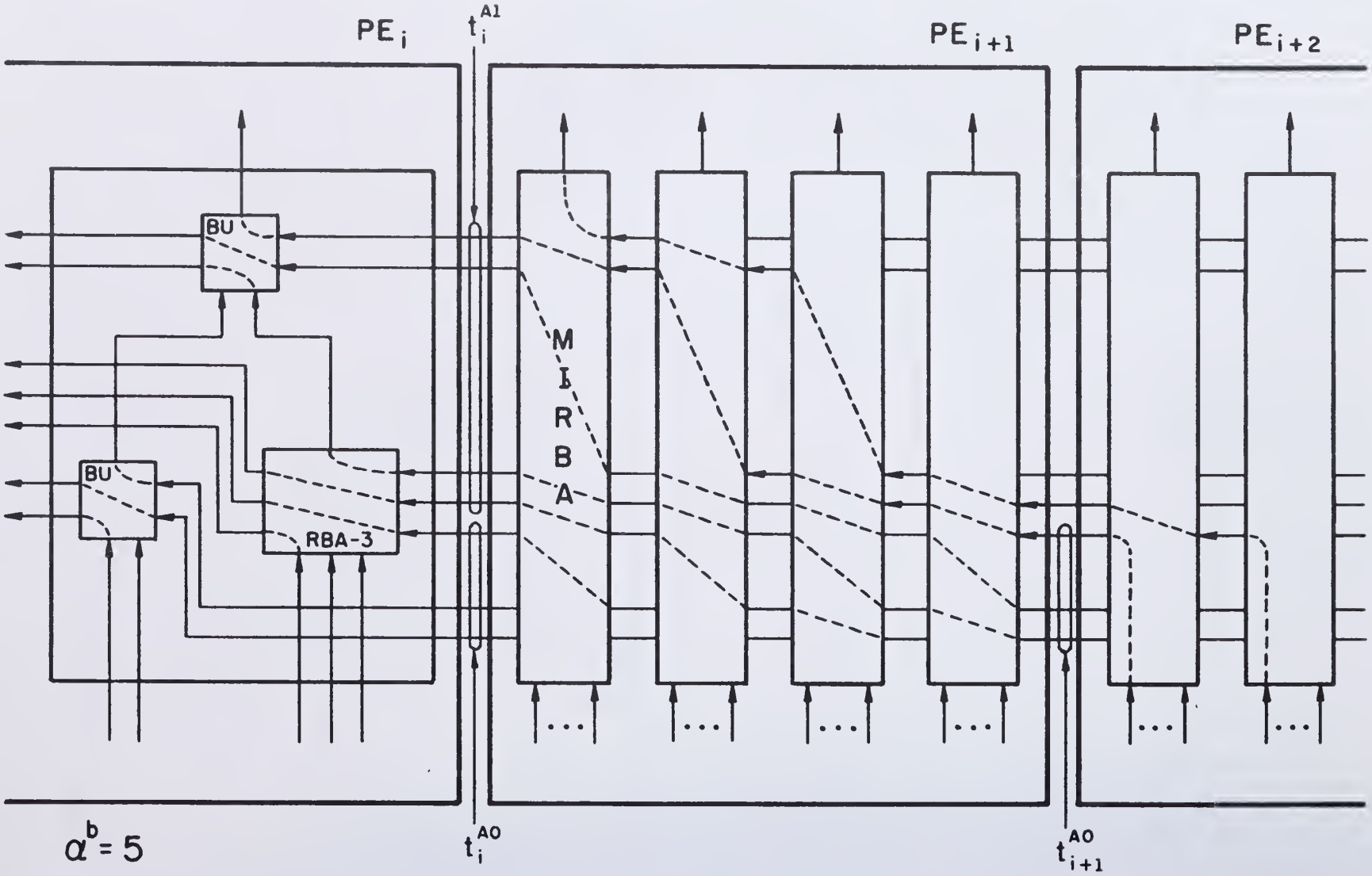


Figure 3.13 Illustration of Tree Structure for a MIRBA using RBA-2s and RBA-3s. (k = 4)

The tree structure configurations described in 3.6.2.1.3.2 and 3.6.2.1.3.3 have the following advantages compared to Rohatsch's technique.

a) It is more general and has the same configuration for any value of k.

b) It makes use of only one kind of cell, that is, Borovec Unit for the implementation of MIRBA.

c) The various BUs are uniformly and regularly interconnected.

Because of b) and c) above, this implementation meets the LSI constraints of structure regularity and minimum cell number type.

In terms of our notation of Section 2.4,

$$FMA^{F}_{i} = FMA^{F}_{i-1} \quad \forall\ i,\ 1 \leq i \leq n$$

$$FMA^{F}_{0} = m_j$$

where FMA^{F}_{i} = modifier value which is sent by PE_i along with microinstruction FMA to PE_{i+1}.

FMA^{F}_{0} = modifier value sent by MCU along with microinstruction FMA to PE_i.

m_j = Multiplier (or Quotient) digit.

$$FMA^{G}_{i+1} = \{t^{P}_{i},\ t^{A}_{i}\}$$

where t^{P}_{i} = Product Transfer to PE_i from PE_{i+1}.

t^{A}_{i} = MIRBA Output Transfer from PE_{i+1}.

α_{FMA} = 2.

3.6.3 Multi-sum (MS) microinstruction - This microinstruction forms the sum of N digit vectors where N is the number of inputs of a MIRBA used in the implementation of microinstruction FMA. N depends on the digit algorithm used for FMA. In any case $N \leq k+1$.

The digit level transfer function is given by

$$a'_i = x_i^1 + x_i^2 + \ldots + x_i^N - r\, t_{i-1}^A + t_i^A \qquad (3.6.13)$$

where $a'_i , x_i \in \{(\overline{r-1}),(\overline{r-2}),\ldots,\bar{1},0,1,\ldots,(r-1)\}$

If we designate the set of arithmetic transformations performed by a Borovec Unit as Borovec Unit Transformation (BAT), then the transfer function in Equation (3.6.13) is realized by a series of $\lceil \log_2 N \rceil$ BATs. This is discussed earlier in the design of a MIRBA in Section 3.6.2.1.3.

Implementation

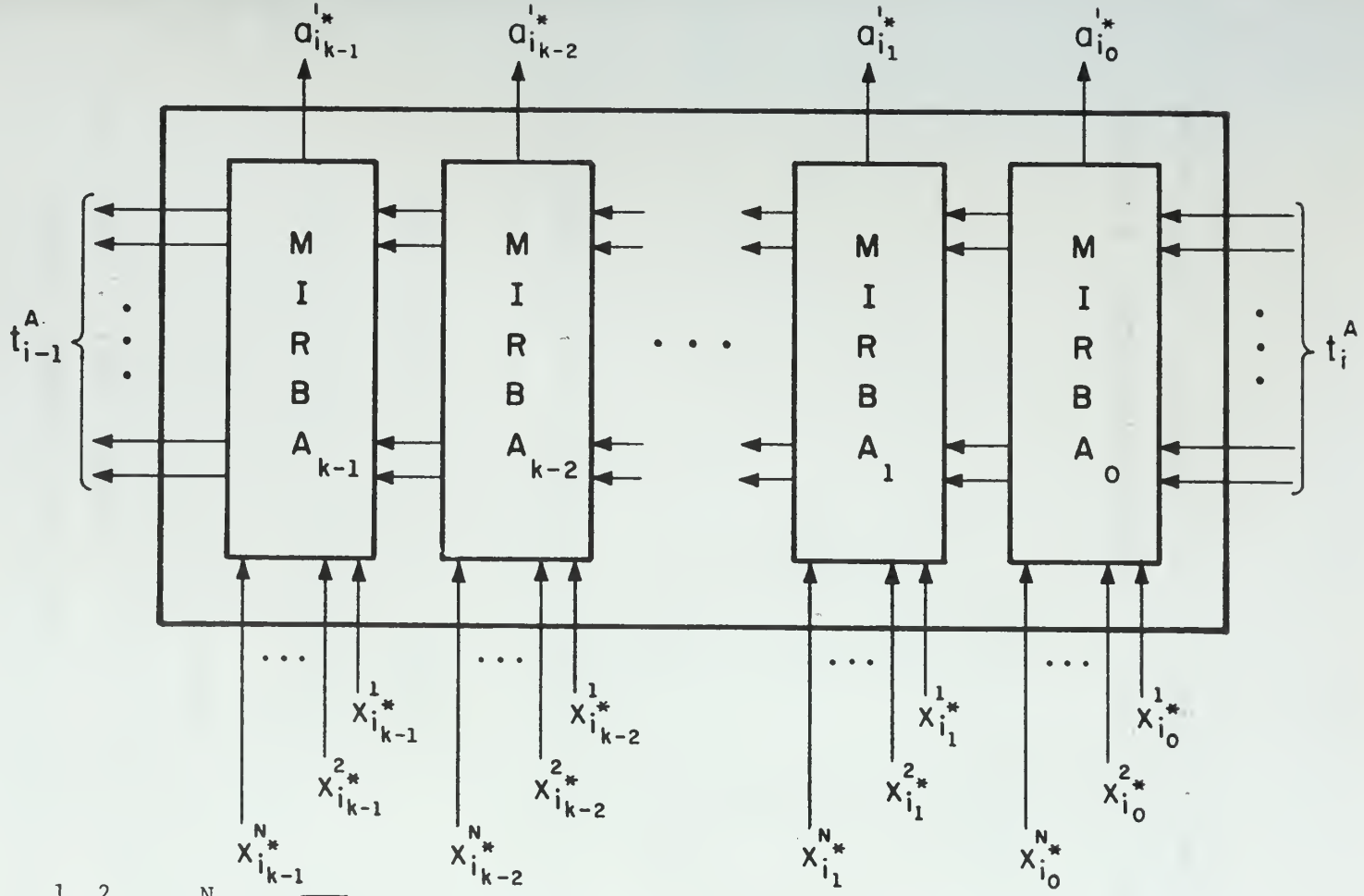
The MS digit algorithm can be implemented by making use of MIRBAs already existing in the digit processing logic of the processing element. This is shown in Figure 3.14.

The MS microinstruction can be represented in the notation of Section 2.4 as follows.

$$MS^F_i = MS^F_{i-1} \quad , \forall i \quad 1 \leq i \leq n$$

$$MS^F_0 = \langle \text{Null} \rangle$$

$$MS^G_{i+1} = t_i^A$$



$a'_i, x^1_i, x^2_i, \ldots, x^N_i \in \{(\overline{r-1}), (\overline{r-2}), \ldots, \bar{1}, 0, 1, \ldots, (r-2), (r-1)\}$

$t^A_i, t^A_{i-1} \in \{(\overline{N-1}), (\overline{N-2}), \ldots, \bar{1}, 0, 1, \ldots, (N-2), (N-1)\}$

$N \leq k + 1$

Figure 3.14 Illustration of Digit Algorithm for Microinstruction MS.

and $\alpha_{MS} = \left|\frac{\alpha^b}{k}\right|$

$= 2$ for radix-$2^k < 32$

$= 1$ for radix ≥ 32.

3.6.4 Normalize Recode (NR) microinstruction - This microinstruction is used to normalize an operand according to Definition 3 given in Section 3.5.1.

Given an operand of the form

$$X \equiv . x_1 x_2 \cdots x_i \cdots x_j x_{j+1} \cdots x_n$$

such that

$$|x_i| > 0 \qquad 1 \leq i \leq j-1,$$

$$x_j = 0,$$

and $$|x_i| \geq 0 \qquad j+1 \leq i \leq n$$

the NR microinstruction transforms the operand X into an algebraically equivalent operand X´

$$X' \equiv . 00 \ldots 0\, x'_h x'_{h+1} \cdots x'_i \cdots x'_j \cdot x'_{j+1} \cdots x'_n$$

where sign (x'_i) = sign (x_1), $h \leq i \leq j-1$ $\qquad h \geq 1$

$$x'_j = x_j = 0$$

and $$x'_k = x_k, \quad j+1 \leq k \leq n$$

For example, if radix r=10, then the numbers $.1\bar{9}\bar{9}70\bar{4}$, $.1\bar{9}0\bar{9}704$, $.17\bar{9}\bar{4}\bar{1}\bar{2}$ and $.10\bar{9}01\bar{8}$ would be recoded respectively as $.001704$, $.010\bar{9}704$, $.160608$ and $.10\bar{9}01\bar{8}$.

Digit Algorithm

Let

S_{OP} = Sign of the operand

S_i = Sign of digit x_i = $\begin{cases} 0 & \text{if +ve} \\ 1 & \text{if -ve} \end{cases}$

r = radix

$|x_i|$ = magnitude of digit x_i

The digit algorithm is given by the flowchart of Figure 3.15. Initially S_{OP} is known and is equal to S_1.

In terms of the notation of Section 2.4,

$$_{NR}F_i = S_{OP} \qquad 1 \leq i \leq j - 2$$

$$= \text{<Null>} \qquad i \geq j - 1$$

where

$$S_{OP} = S_1 \quad \text{for} \quad i \geq 0.$$

j is the index of first zero digit in the operand d-vector.

$$_{NR}G_{i+1} = x^{\dagger}_{i+1} \qquad j \leq i \leq j-1$$

and $\alpha_{NR} = 1.$

3.6.5 Assimilation Recode (AR) microinstruction - This microinstruction is used to assimilate (convert) a signed-digit redundant operand into an algebraically equivalent operand such that all the digits in the recoded form are of the same sign as the sign of original operand.

---

†In the actual implementation of the digit algorithm for NR, $_{NR}G_{i+1}$ information consists of S_{i+1} and Z_{i+1} where S_{i+1} is a bit carrying sign information of digit x_{i+1} and Z_{i+1} is also a bit whose two states indicate where digit x_{i+1} is zero or not. (cf. Section 4.3.2.3.6.2)

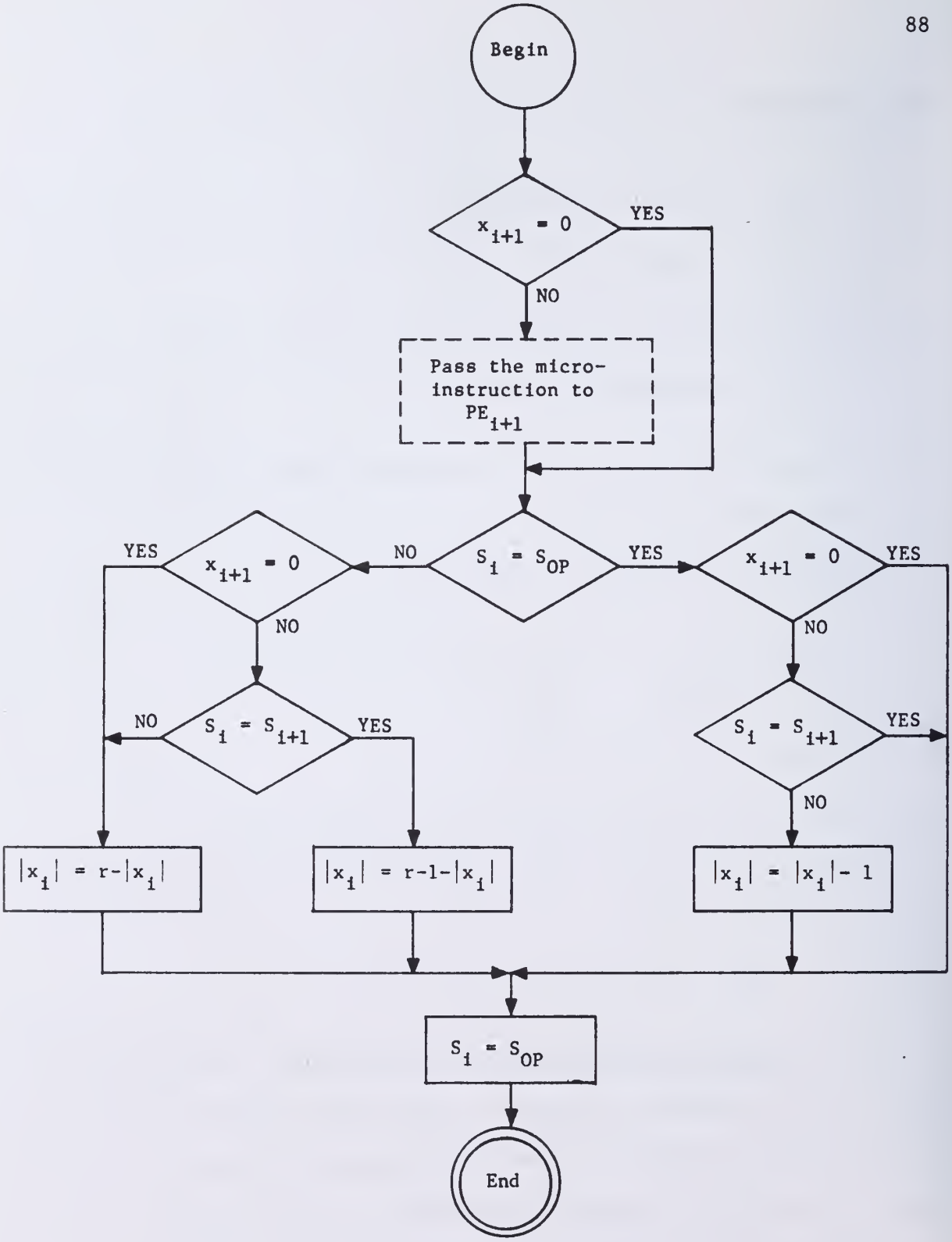


Figure 3.15 Flowchart of the Digit Algorithm for Microinstruction NR.

Given a signed-digit operand of the form

$$X \equiv . x_1 x_2 \ldots x_i 0 x_{i+2} \ldots x_k 00 \ldots 0 x_{k+\ell} \ldots x_n$$

such that

$$\text{sign } (x_{i+1}) = \text{sign } (x_{i+2}) \text{ and}$$

$$\text{sign } (x_{k+1}) = \text{sign } (x_{k+2}) = \ldots = \text{sign } (x_{k+\ell-1}) = \text{sign } (x_{k+\ell})$$

That is, all the zero digits in the d-vector of number X have the same sign as that of the first nonzero digit to the immediate right of a string of zeros (of length ≥ 1), the AR digit algorithm would recode X into X´

$$X' \equiv . x_1' x_2' \ldots x_i' x_{i+1}' x_{i+2}' \ldots x_k' x_{k+1}' \ldots x_{k+\ell}' \ldots x_n'$$

such that

$$X = X' \text{ and}$$

$$\text{sign } (x_i') = \text{sign } (x_{i+1}) = \text{sign } (x) \quad \forall\, i,\ 1 \leq i \leq n$$

Digit Algorithm

Let S_{OP} = sign of the operand

S_i = sign of the digit i

S_{i+1} = sign of the digit i+1

x_i = digit i of operand X

The digit algorithm is almost identical with that for NR microinstruction except that the microinstruction acts on all the digits of the operand. It is given by the flowchart shown in Figure 3.16.

In the notation of Section 2.4,

$$AR^F{}_i = S_{OP} \qquad 1 \leq i \leq n$$

$$= S_1$$

$$AR^G{}_{i+1} = S_{i+1} \qquad 1 \leq i \leq n$$

α_{AR} = Variable depending on the d-vector

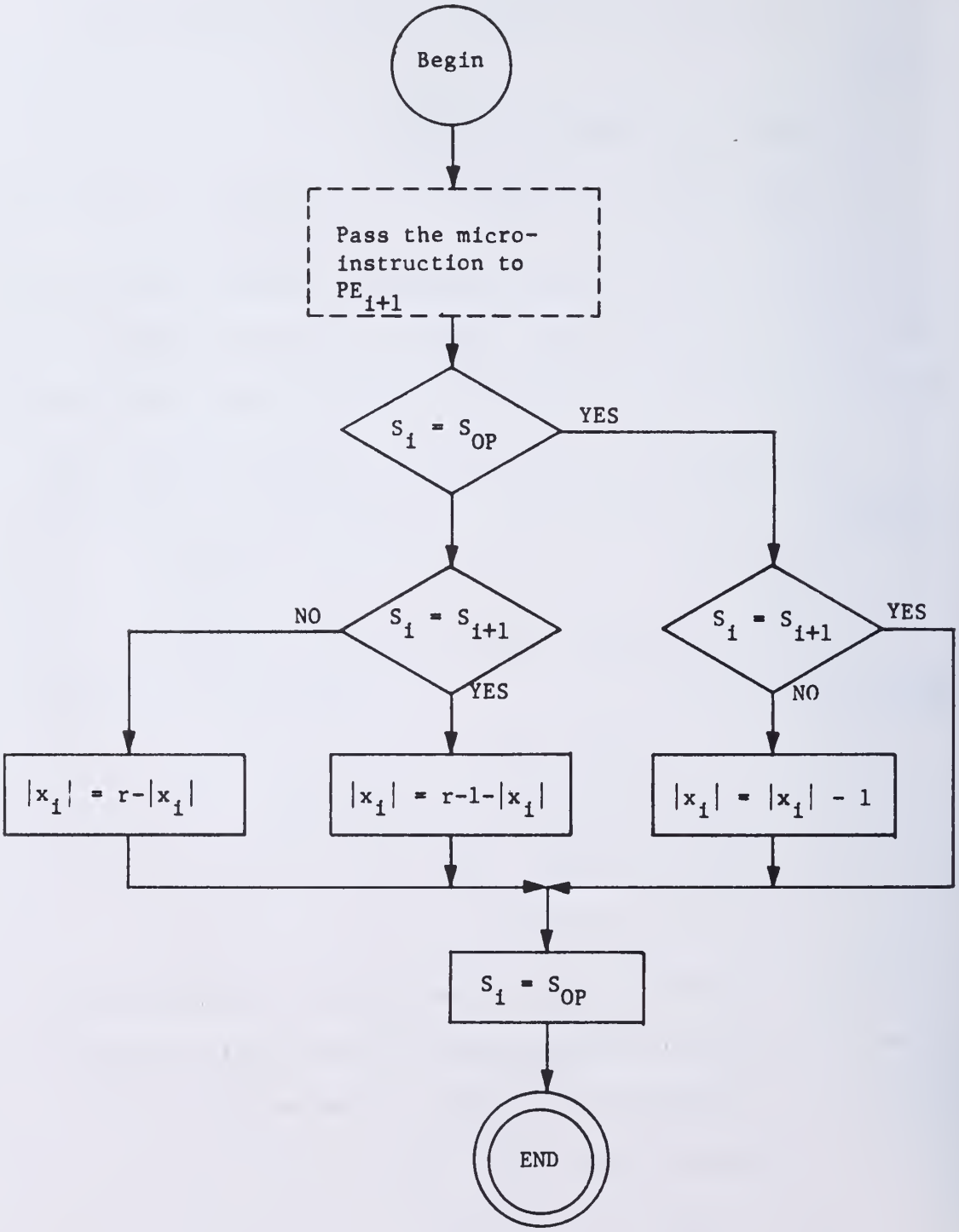


Figure 3.16 Flowchart of the Digit Algorithm for Microinstruction AR.

4. LOGIC DESIGN OF THE PROCESSING ELEMENT

4.1 Introduction

In this chapter, the logic design of the Processing Element (PE) is developed and discussed in detail. The major components of the PE are the Register File for the temporary storage of active operands, the Digit Processing Logic (DPL) which is essentially a large combinational logic circuit and the Processing Element Control Logic (PCL) which supplies the control signals in proper temporal order to condition the combinational DPL to execute the various microinstructions. The major considerations in the logic design of the PE are the LSI technology constraints: namely, the PE should require as few external pins as possible and that the logical organization of the PE should have structural uniformity and regularity.

Section 4.2 discusses the logic design of data path structure of the PE and in Section 4.3 is given the logical organization and detailed design of the control algorithms for the generation of control signals. Finally, Section 4.4 discusses the logic complexity of the DPL and the PE control logic in terms of the number of gates and the external pins for the PE module as a function of the bit width of the PE module.

4.2 Block Diagram Description of a Processing Element

Figure 4.1 shows the schematic block diagram of a Processing Element. It consists of three main components--Digit Processing Logic (DPL), Register File and Control. The Register file comprises a set of digit-wide registers which are used to hold the operand digits and

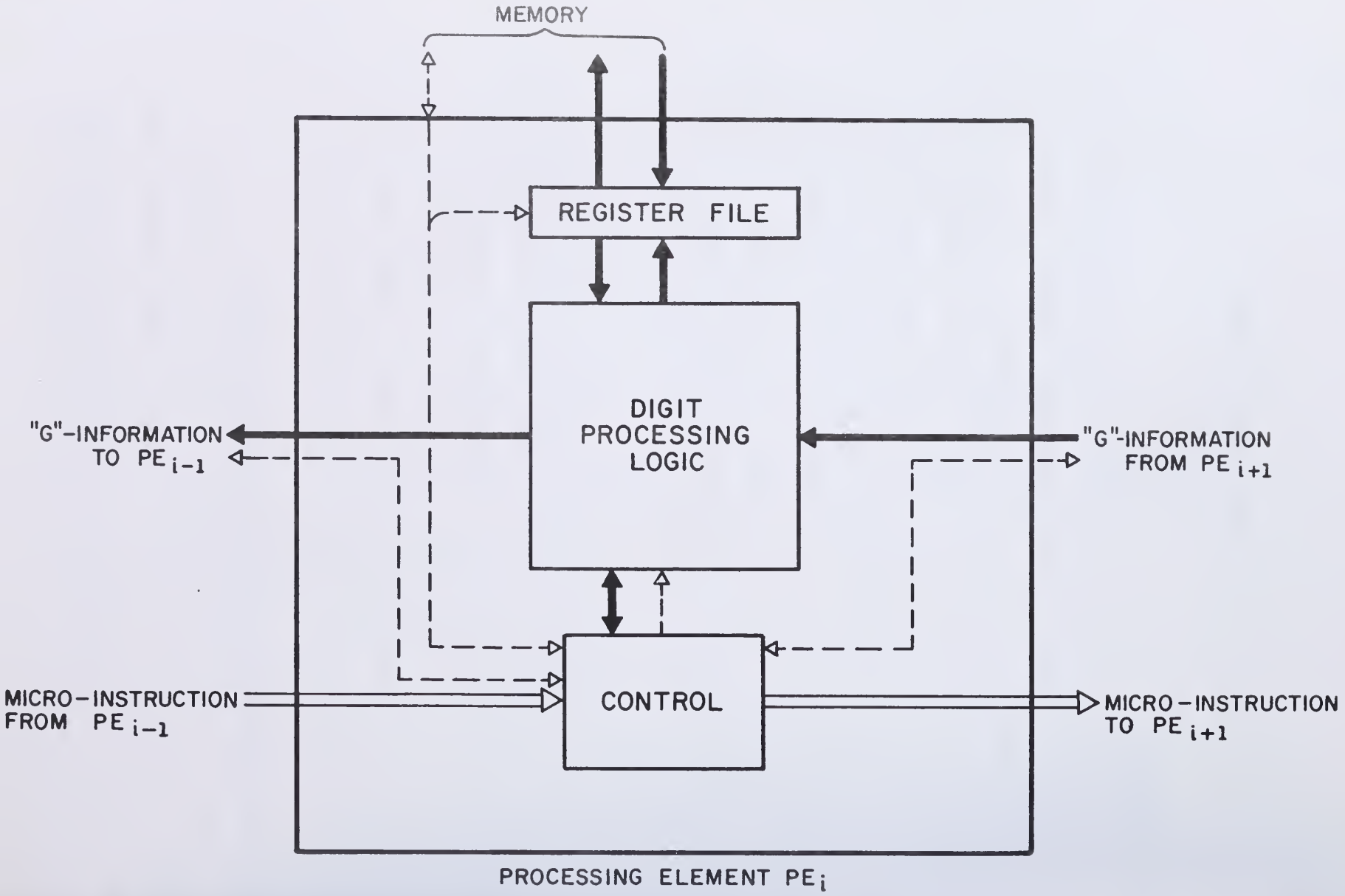


Figure 4.1 Block Diagram of a Processing Element.

result digits. The registers could also be used to hold intermediate result digits temporarily. Inter-register transfer microinstructions operate on these registers. The Digit Processing Logic is essentially combinational processing logic (along with some storage for G-information) and is used to process the microinstructions of the PE. The DPL operates on the operand digits stored in the register file of the PE and G-information from its right neighboring PEs. It also generates the G-information for its left neighbor PEs. The Control issues the timing control signals to the processing logic for sequencing the various steps of the digit algorithms for the microinstructions. It also coordinates the actions of the PEs by accepting the microinstructions and G-information from neighboring PEs and by transmitting the microinstructions to the right neighbor and generating G-information for its left neighbor PE.

4.2.1 Register file - The register file is a set of registers that are used to hold the operand and result digits. Each PE retains one digit of each of the active operands. Each register is (k+1) bits long to hold the k-magnitude bits and one sign bit of one SM_r-encoded radix-2^k digit. There must be at least three registers in a PE: an accumulator register, a multiplier-quotient (MQ) register, and an operand register. However, the multi-sum microinstruction MS requires N operands and hence N storage registers where N represents the number of operands the radix-2^k adder in the DPL is capable of adding simultaneously. It was shown earlier in Sections 3.6.2.1.1 and 3.6.2.1.2 that the radix-2^k adder adds

either (k+1) or 3 operand digits depending on how the algorithm for microinstruction FMA is implemented. For the present discussion, we shall assume that the adder is capable of adding (k+1) operand digits simultaneously and that k = 4. The register file would thus contain at least five registers. Additional registers can be added to the register file. One possible use of such registers is to hold the intermediate results which are needed so soon after they are calculated that storing them and retrieving them from memory would unnecessarily delay the processing. The number of desirable intermediate result registers is determined by the method of communicating between memory and the arithmetic unit, the number of extra pins, if any, required for the identification address of these registers and their contribution to the overall logic complexity of the chip. Figure 4.2 shows an internal register file containing five registers INR1, INR2,...,INR5. In this thesis, registers INR1, INR2 and INR3 act respectively as Accumulator, operand register and MQ-register.

The registers in the register file are loaded from a buffer register, IBR whose contents are determined by the Internal Register Input Bus Selector, sRIB in the Digit Processing Logic (discussed later in Section 4.2.2.7.3). Similarly, the contents of the registers are inputed to the DPL either directly or through an Output Bus Selector sROB, also in the DPL. The control signals gINR[x], x = 1,2,...,5 for loading the registers are provided by the local control in the PE.

Because of the bus mechanism for inputing and outputing of data in the register file, any whole operand register (consisting of corresponding operand digit registers distributed in the various PEs) can be used

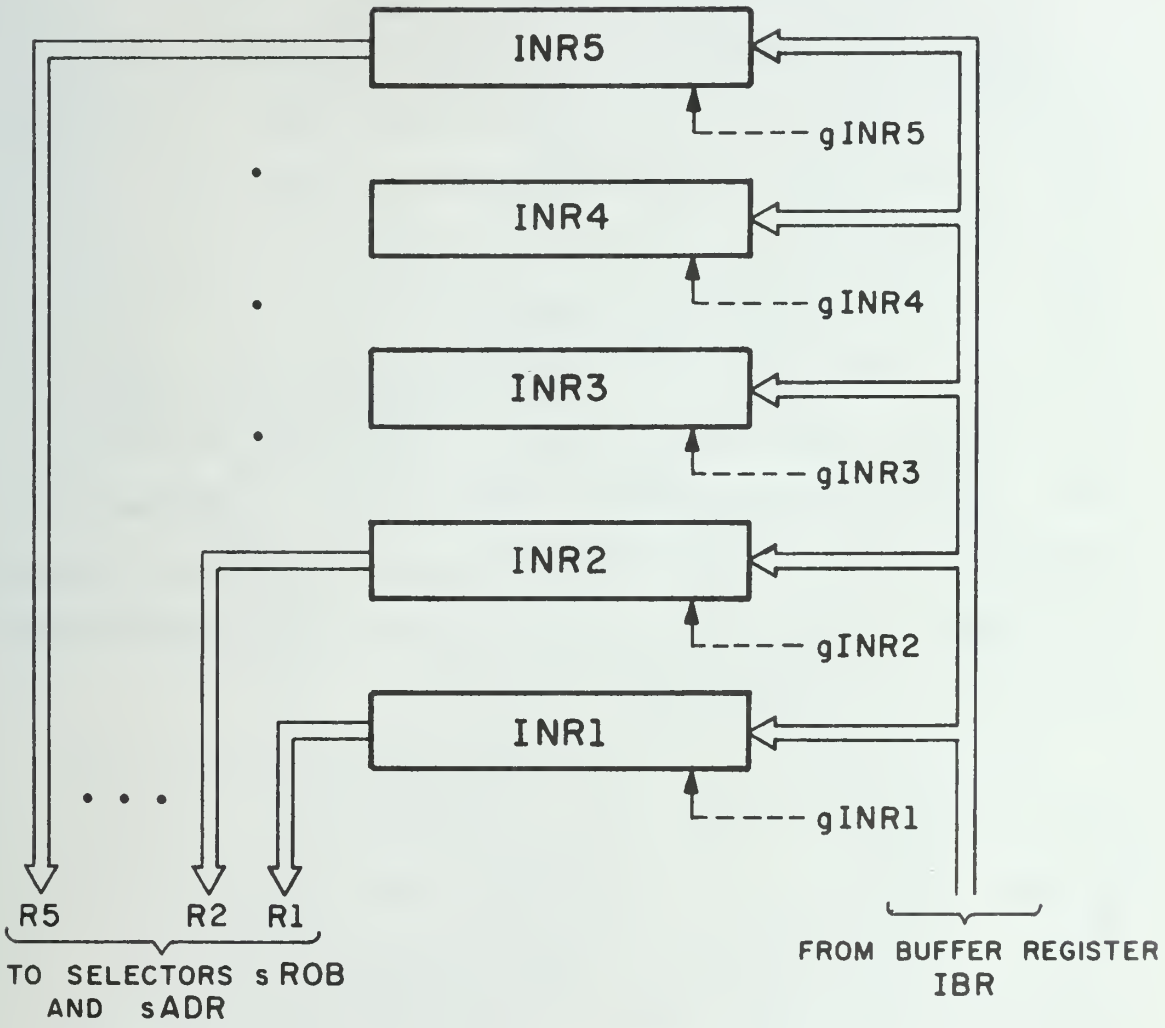


Figure 4.2 Block Diagram of the Register File of the PE.

as a shift register, capable of shifting one digit at a time. This is made use of in microinstructions LS and RS. R1, R2,...,R5 denote the outputs (contents) of registers INR1, INR2,...,INR5, respectively.

4.2.2 Logic design of digit processing logic

4.2.2.1 Block diagram description of DPL - Figure 4.3 shows the data flow structure of the Digit Processing Logic (DPL) in block diagram form. It consists of three major components--the Digit Product Generator, DPG, a radix-2^k multi-input adder MIAD, and a Digit Sum Encoder, DSE. In addition to these three main components, there are selector networks sADR for the adder input, sDSE for the Sum Encoder, sROB for selecting the output of internal registers in the register file and sRIB for selecting the inputs to the in-bus of the register file and sTOP for selecting the contents of 'Transfer' Output Port (TOP). Besides, there are two registers GIR and APR for storing the G-information and multiplicand digit† from the adjacent right neighbor PE.

The Digit Product Generator, DPG forms the product array in redundant binary form, of two SM_r-encoded radix-2^k digits m_j and ϕ_i. The digit ϕ_i comes from the operand register INR2 via the output bus selector sROB and the multiplier digit m_j is inputed to the DPG, from the microinstruction register MIR in local control logic of the PE.

The multi-input adder, MIAD adds the w_i columns of the redundant binary product array formed by DPG and the collective product transfer

†The necessity of the register APR for the multiplicand digit from the adjacent PE would be clear from the discussion in Section 4.2.2.5.

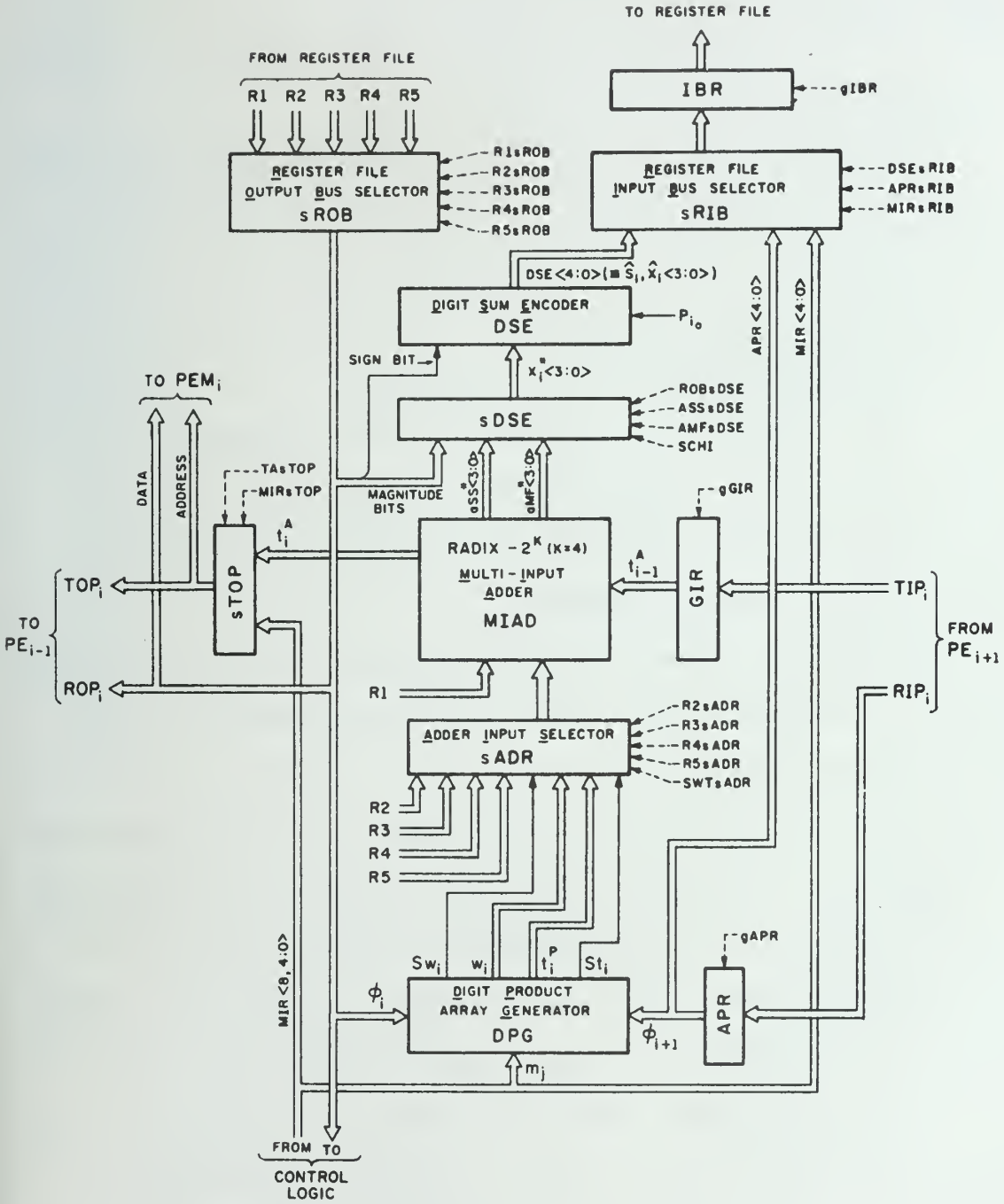


Figure 4.3 Block Diagram of Digit Processing Logic (DPL).

t_i^P. The MIAD is also used to add the two operand digits a_i^* and ϕ_i^* from the registers R1 and R2 for the microinstruction SS and to add the operand digits for the microinstruction MS. The radix-2^k multi-input adder is made up of k-stages of MIRBAs.

The Digit Sum Encoder DSE converts the redundant binary sum output of adder MIAD to the SM_r format for local storage in the accumulator register INR1 of the register file or transfer out of the PE. The DSE is also used in the microinstructions AR and NR for forming the radix and diminished radix complement of the magnitude bits of the accumulator register INR1 and also for subtracting unity from the magnitude of the accumulator contents. In addition, sDSE and DSE are made use of in inter-register transfer microinstructions TD and TI for direct and reversed-sign inter-register transfer.

The Adder Input Selector sADR routes appropriate data in redundant binary form to the MIAD inputs depending on the microinstruction the PE is executing at that time.

The selector sDSE selects the appropriate input to the encoder DSE.

Also shown in the Figure 4.3 are input and output ports designated as TIP_i, RIP_i and TOP_i, ROP_i, respectively. The input port TIP_i and RIP_i, respectively carry the 'transfer' (carry or borrow) from adjacent MIAD and the contents of some register in the register file of the adjacent PE_{i+1}. These ports essentially carry the 'G-information' from the adjacent PE_{i+1} for the microinstruction that is being executed by the present PE_i. The output ports TOP_i and ROP_i are, however, shared to carry the 'G-information' for the left neighbor PE_{i-1} and also the address and data information respectively for the local operand memory

PEM_i. This is made use of, for fetching data from and storing data to the PEM_i under the control of microinstructions LPM and SPM.

The selector sTOP selects either the 'transfer' information from MIAD or the address bits and Read/Write bit for PEM_i.

The details of the logic design of the various blocks described above are discussed next in the following sections. Since the logic complexity of the major components DPG, MIAD and DSE and sADR are dependent on the choice of logic vector encoding for the redundant binary digits, the three logic vector encodings considered for study are described first. It is followed by the logic design details of the major components.

4.2.2.2 Choice of logic vector encodings - As mentioned in Section 3.4.1, the redundant binary (RB_r) mode of encoding for a radix-r signed digit is used for the arithmetic processing. Each redundant binary digit requires 2 bits for representation. There are nine distinct ways under permutation and negation [40], of assigning three values ($\bar{1}$,0,1) to four states of two binary logic variables. Of the nine ways, three encodings were chosen for this study because they are the simplest as far as the conversion from the SM_r mode to the chosen encoding for RB_r mode is concerned. Let a radix-2^k, signed digit $\hat{x}_i$, encoded in SM_r mode, be represented by a k+1-tuple $(S_i,\hat{x}_{i_{k-1}},\hat{x}_{i_{k-2}},\ldots,\hat{x}_{i_0})$ such that

$$\hat{x}_i = (-1)^{S_i} \sum_{j=0}^{k-1} \hat{x}_{i_j} \cdot 2^j \qquad \hat{x}_{i_j} \in \{0,1\}.$$

The corresponding RB_r encoded form is given by

$$\hat{x}_i = \sum_{j=0}^{k-1} x^*_{i_j} \cdot 2^j \qquad x^*_{i_j} \in \{\bar{1},0,1\}.$$

Let the redundant binary digit $x^*_{i_j}$ be represented by a 2-tuple logic vector (χ_{i_j}, x_{i_j}) where

$$\chi_{i_j}, x_{i_j} \in \{0,1\}.$$

The three logic vector encodings for the redundant binary digit $x^*_{i_j}$ considered in this research are given in Table 4.1.

Table 4.1

Logic Vector Encodings

Binary 2-tuple logic vector		Encodings		
		LVE_1	LVE_2	LVE_3
χ_{i_j}	x_{i_j}	$x^*_{i_j}$	$x^*_{i_j}$	$x^*_{i_j}$
0	0	0	0	0
0	1	$\bar{1}$	1	1
1	0	1	D.C	0
1	1	0	$\bar{1}$	$\bar{1}$

The conversion from SM_r mode to the encoding format LVE_3 is the simplest and is equivalent to attaching the sign S_i of the SM_r encoded radix-2^k digit to each magnitude bit $\hat{x}_{i_j}$ individually. The conversion for the three encodings are given by

$$\underline{LVE_1}: \quad x^*_{i_j} = \chi_{i_j} - x_{i_j}$$

$$x_{i_j} = \hat{x}_{i_j} \oplus S_i \qquad (4.1)$$

$$\chi_{i_j} = S_i$$

where $\oplus$ stands for exclusive--OR

or

$$x_{i_j} = S_i \wedge \hat{x}_{i_j}$$

$$\chi_{i_j} = \overline{S_i} \wedge \hat{x}_{i_j} \qquad (4.2)$$

$$\underline{LVE_1}: \quad x^*_{i_j} = x_{i_j} - 2x_{i_j} \text{ with } x_{i_j}\ \bar{x}_{i_j} \text{ disallowed}$$

$$x_{i_j} = \hat{x}_{i_j}$$

$$\chi_{i_j} = S_i \wedge \hat{x}_{i_j} \qquad (4.3)$$

$$\underline{LVE_3}: \quad x^*_{i_j} = (-1)^{\chi_{i_j}} \cdot x_{i_j}$$

$$x_{i_j} = \hat{x}_{i_j} \qquad (4.4)$$

$$\chi_{i_j} = S_i$$

For the encodings LVE_1 and LVE_2, the conversion logic requires one exclusive-OR gate (Equation(4.1)) or two AND-gates (Equation (4.2)), and one AND-gate (Equation (4.3)) for each redundant binary digit respectively. For one radix-2^k digit conversion, there are k redundant binary digits.

Encoding LVE_3 is essentially a sign and magnitude encoding of the redundant binary digit $x^*_{i_j}$ by the 2-tuple (χ_{i_j}, x_{i_j}). Logic variables χ_{i_j}

and x_{i_j} respectively act as sign and magnitude bits. This encoding format would also be referred to, in subsequent discussion, as SM_b format where subscript b indicates radix-2 (or binary).

4.2.2.3 Logic design of RBA-2 (BU) - Let ℓ_ν^*, m_ν^* denote the redundant binary inputs and d_ν^* denote the redundant binary output of a RBA-2. Further let ℓ_ν^*, m_ν^* and d_ν^* be respectively represented by the logic variable pairs (λ_ν, ℓ_ν), (μ_ν, m_ν) and (δ_ν, d_ν). Also let t_ν^+, t_ν^- and $t_{\nu-1}^+$, $t_{\nu-1}^-$ be the input and output 'Transfers' of the RBA-2 as shown in Figure 4.4. In the configuration shown in Figure 4.4, it has a cascade combination of a symmetric subtracter and a symmetric adder. Robertson [40] has given the logic equations for the symmetric subtracter and symmetric adder for all the nine distinct encodings referred earlier. Using those results, the logic equations for the RBA-2 for the three logic vector encodings being considered here are given as follows:

$$
\begin{aligned}
\underline{LVE_1}: \quad d_\nu &= \lambda_\nu \oplus \ell_\nu \oplus \mu_\nu \oplus m_\nu \oplus t_\nu^- \\
\delta_\nu &= t_\nu^+ \\
t_{\nu-1}^- &= \ell_\nu m_\nu \vee \bar{\lambda}_\nu (\ell_\nu \vee m_\nu) \\
t_{\nu-1}^+ &= w_\nu \mu_\nu \vee \bar{\bar{t}}_\nu^- (w_\nu \vee \mu_\nu) \\
w_\nu &= \lambda_\nu \oplus \ell_\nu \oplus m_\nu
\end{aligned}
\qquad (4.5)
$$

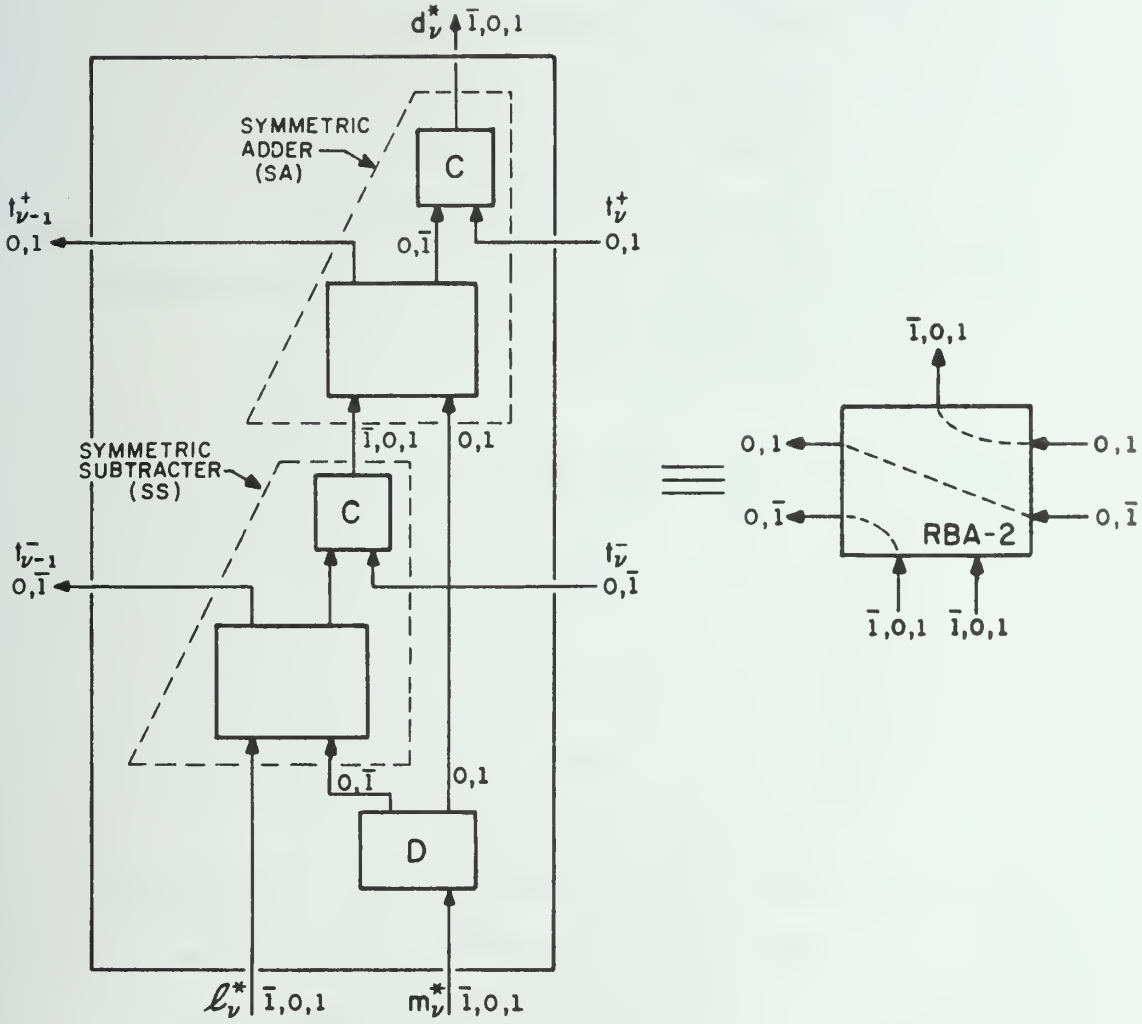


Figure 4.4 Algebraic Design of a 2-input Redundant Binary Adder (RBA-2).

This is schematically represented in Figure 4.5. Each box in the figure essentially represents a full adder with a slightly modified carry function. Figure 4.6 shows the logic implementation. This implementation requires 22 two input NAND gates and the output digit d_ν^* is available after 12 gate delays. Figure 4.7 shows another logic implementation that requires 27 gates but the output digit is available after only 9 gate delays. Note further that the logic in Figure 4.7 is no longer made of two identical logic substructures. The implementation of Figure 4.6 allows a simpler basic cell for LSI implementation of MIRBA at the cost of larger logic delay.

$$\underline{\text{LVE}_2}: \quad \begin{aligned} d_\nu &= \ell_\nu \oplus m_\nu \oplus t_\nu^+ \oplus t_\nu^- \\ \delta_\nu &= \overline{t_\nu^+}\ (\ell_\nu \oplus t_\nu^- \oplus m_\nu) \\ t_{\nu-1}^- &= \lambda_\nu \vee \overline{\ell_\nu}\ \mu_\nu \\ t_{\nu-1}^+ &= \overline{t_\nu^-}\ ((\ell_\nu \oplus \mu_\nu) \vee \overline{\mu_\nu}\ m_\nu) \vee \overline{\mu_\nu}\ m_\nu\ \ell_\nu\ . \end{aligned} \tag{4.6}$$

The logic implementation of this adder using only 2 input NANDS is shown in Figure 4.8. Thirty-four two input NAND gates are needed. The output digit is available, 13 gate delays after the primary inputs ℓ_ν^* and m_ν^* are stable because the 'Transfer' input t_ν^+ is available 7 gate delay after inputs $\ell_{\nu+1}^*$ and $m_{\nu+1}^*$.

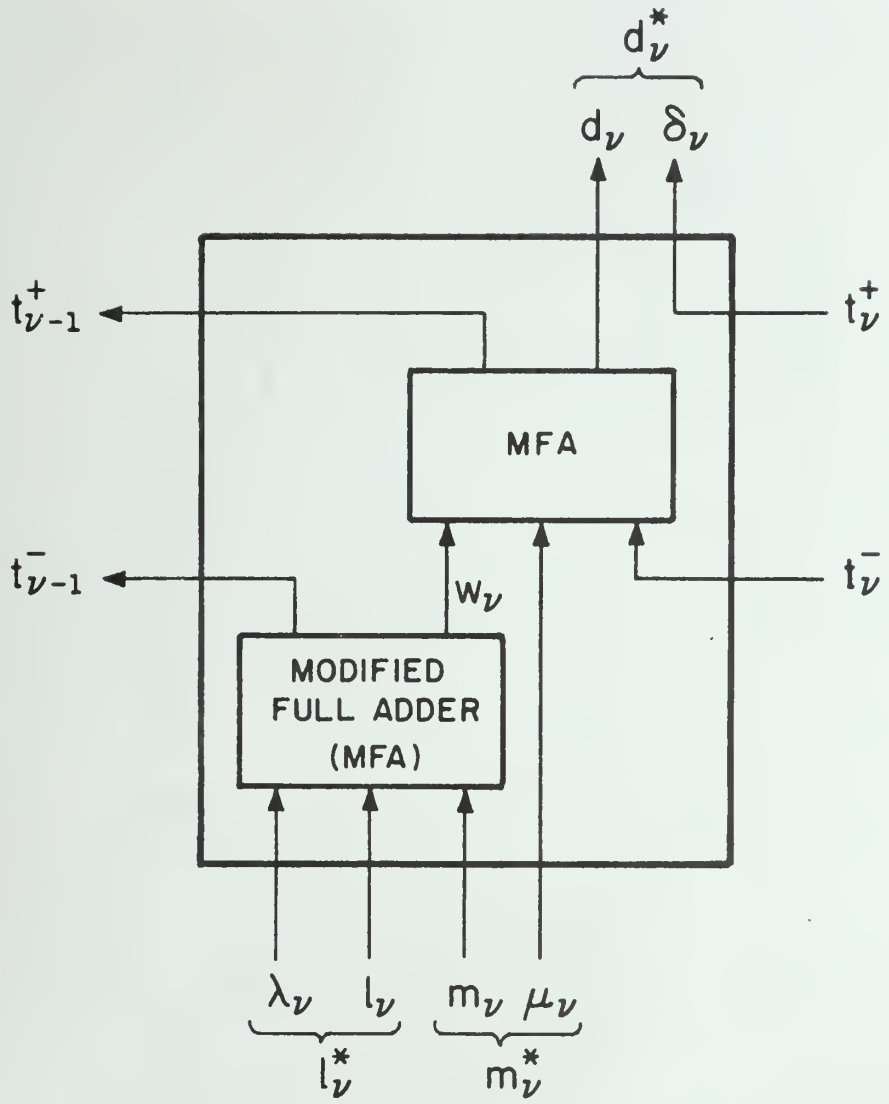


Figure 4.5 Schematic Functional Diagram of an RBA-2 using LVE_1.

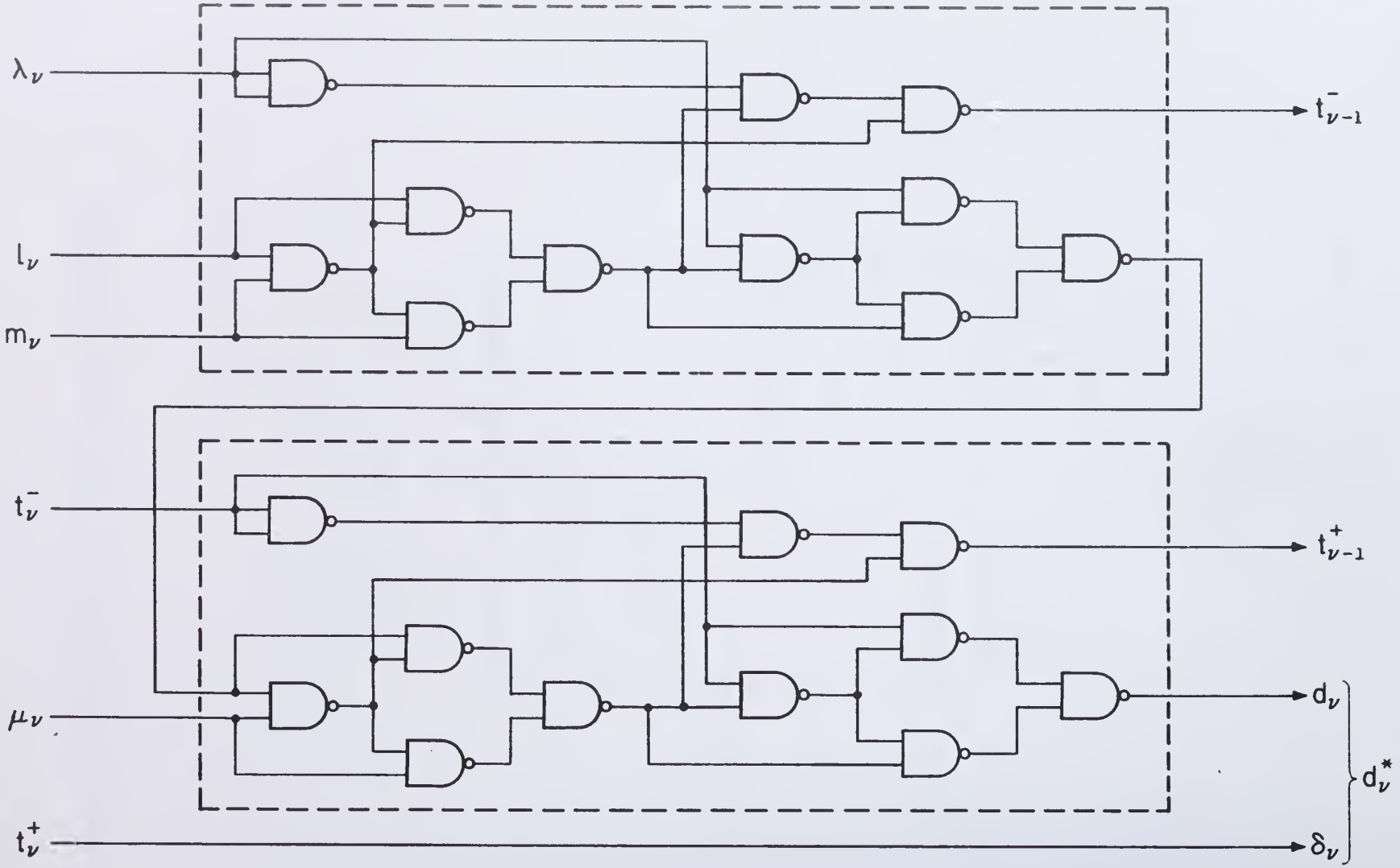


Figure 4.6 Logic Implementation of an RBA-2 using Logic Vector Encoding LVE_1. (Version 1)

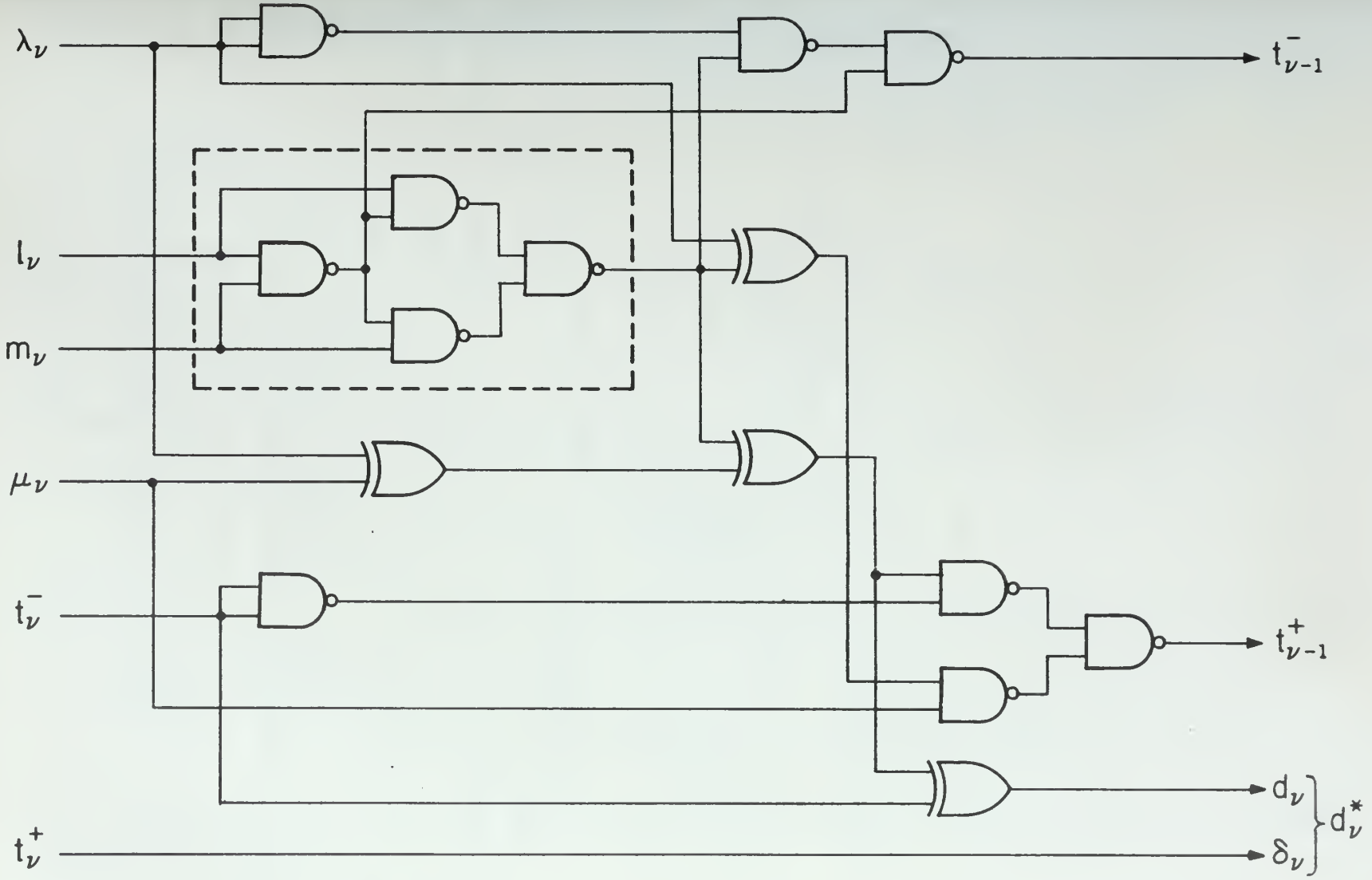


Figure 4.7 Logic Implementation of an RBA-2 using Logic Vector Encoding LVE_1. (Version 2)

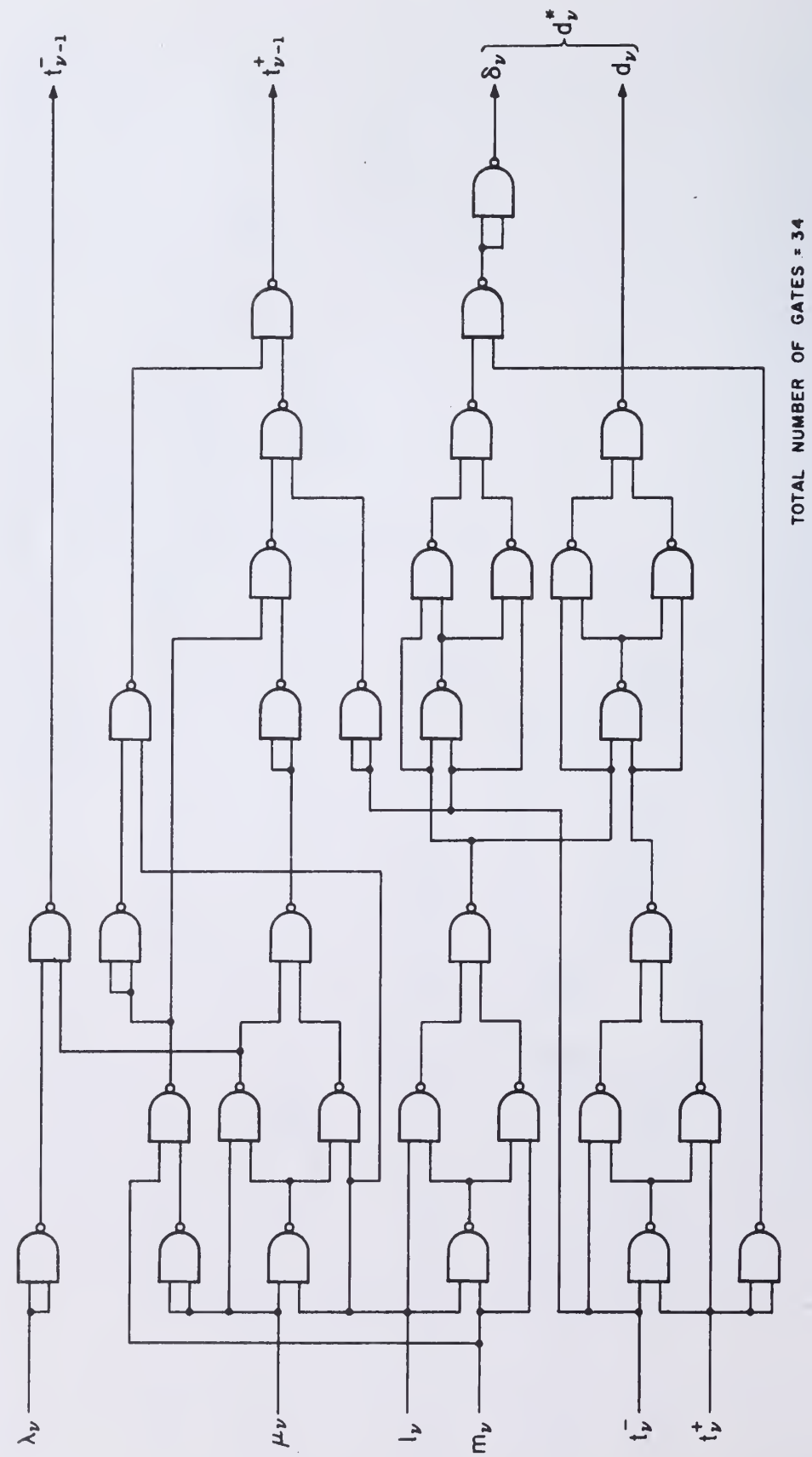


Figure 4.8 Logic Implementation of an RBA-2 using Logic Vector Encoding LVE_2.

$$\underline{LVE_3}: \quad d_\nu = \ell_\nu \oplus m_\nu \oplus t_\nu^+ \oplus t_\nu^-$$

$$\delta_\nu = t_\nu^+ \tag{4.7}$$

$$t_{\nu-1}^- = \lambda_\nu \ell_\nu \vee \mu_\nu m_\nu \overline{\ell_\nu}$$

$$t_{\nu-1}^+ = \overline{t_\nu^-} \ ((\ell_\nu \oplus m_\nu) \vee \ell_\nu \overline{\mu_\nu}) \vee \overline{\mu_\nu} \ell_\nu m_\nu$$

The logic implementation of this adder using only 2 input NANDS is shown in Figure 4.9. This RBA-2 realization requires 26 gates and the output d_ν^* is available 13 gate delays after the primary inputs of this RBA-2 and its adjacent RBA-2 are stable.

Note that the lower gate delay for the sum output of RBA-2 using LVE_1 encoding is achieved because the logic variable d_ν is a function of ℓ_ν^*, m_ν^* and t_ν^- only. In the other two encodings, d_ν is dependent on t_ν^+ also.

4.2.2.4 Logic design of a radix-2^k multi-input adder (MIAD) - The radix-2^k adder MIAD is used for two purposes; 1) to add the columns of the redundant binary product array formed by DPG and 2) to form the sum of the operand digits for microinstructions SS and MS. Figure 4.10 shows the schematic diagram of a radix-16 (k = 4) MIAD. It consists of 4 MIRBAs each of five inputs each. Each MIRBA has two outputs--one a^*MF corresponding to the sum of all the five inputs (used in microinstructions MS and FMA) and a^*SS corresponding to the sum of only two inputs--for microinstruction SS to the left-most BU in the bottom level of the tree of BUs and RBA-3s making up a MIRBA. The proper data is routed to the inputs of MIRBAs by the adder input selector sADR. MATE and MATD are the encoders

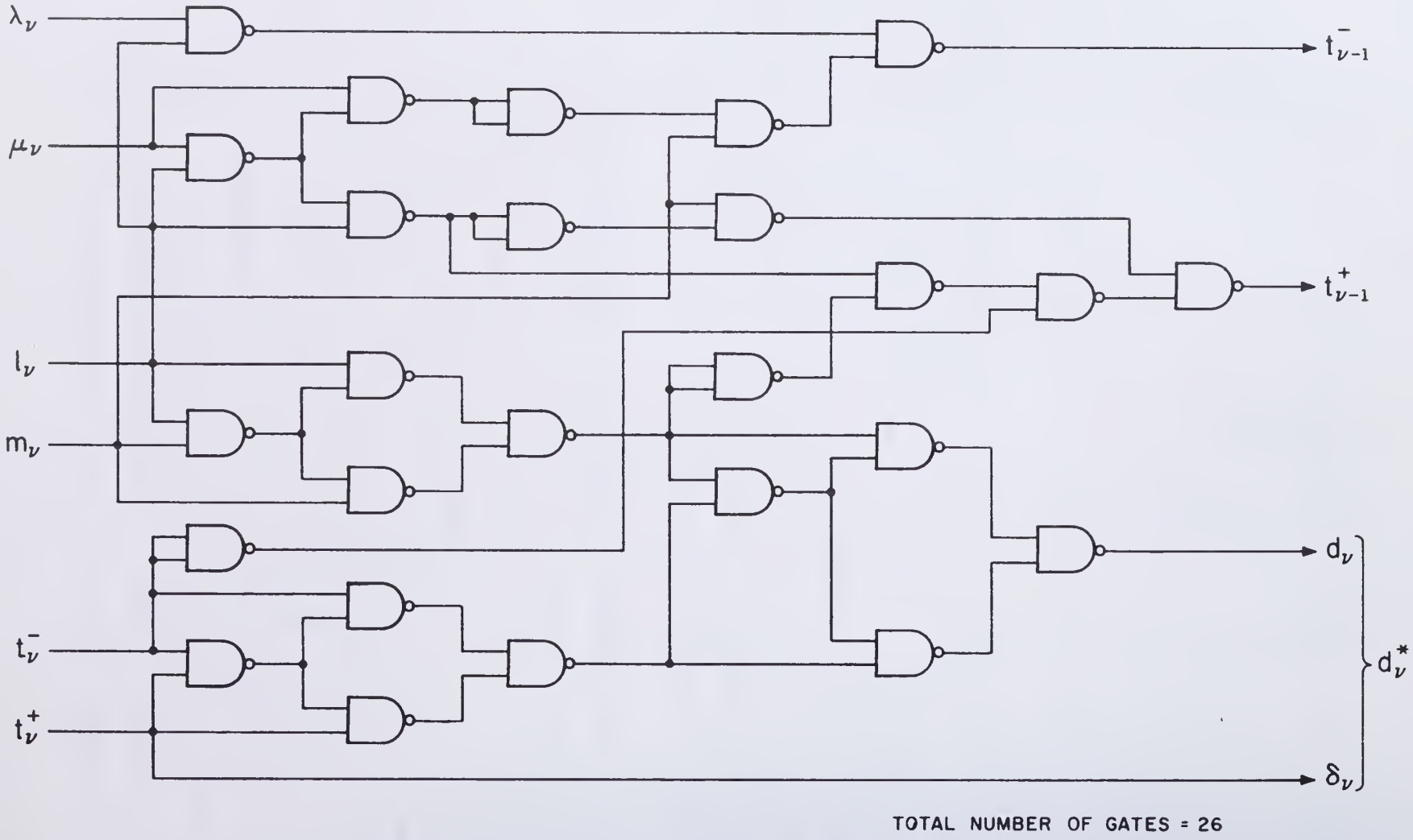
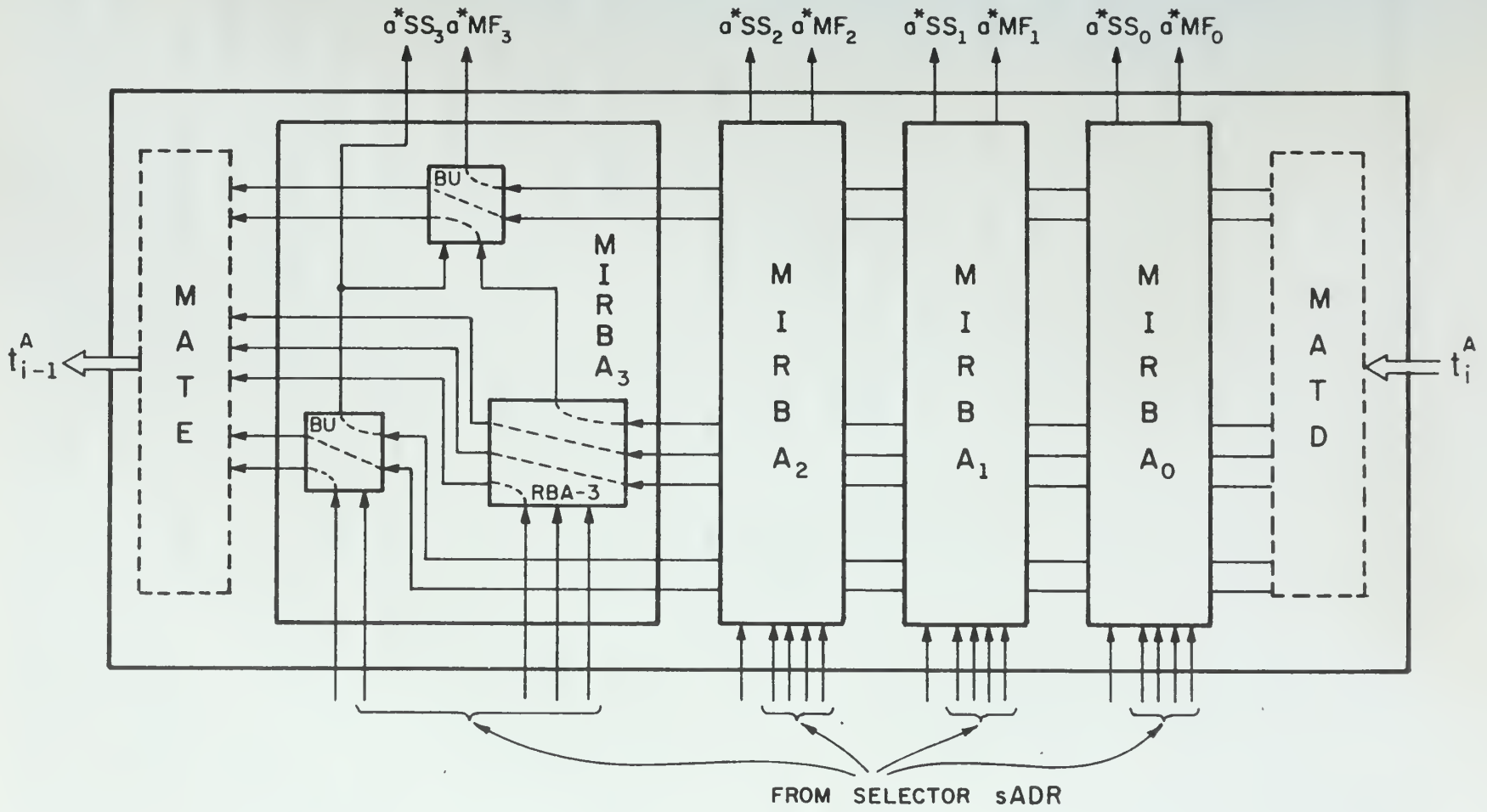


Figure 4.9 Logic Implementation of an RBA-2 using Logic Vector Encoding LVE_3.



Note: MATE and MATD are optional.

Figure 4.10 Schematic Diagram of a Radix-2^k (k=4) Multi-input Adder (MIAD).

and decoders used for reducing the pins required for the 'Adder Transfers' t^A_{i-1} and t^A_i.

In general, a radix-2^k multi-input adder consists of a linear cascade of k MIRBAs. A (k+1) input MIRBA is implemented as a tree structure of RBA-2s and/or RBA-3s. Each MIRBA requires k RBA-2s and are arranged in $L = \lceil \log_2(k+1) \rceil$ levels. Therefore, the total number of gates required for radix-2^k adder is k times the gates needed for each MIRBA and is equal to k^2 times the gates required by each RBA-2 (plus those required by D and C-elements of RBA-3s). Also, the total gate delay for the sum output is L times the delay of each RBA-2 (ignoring the extra delay necessary for control and inter-PE communication of 'Adder Transfers' t^A_i and t^A_{i-1}).

For a (k+1) input adder, the number of pins required for the input and output Adder Transfers t^A_{i-1} and t^A_i are 2k each, and is large for large value of k. One way to reduce the pins necessary for t^A_i and t^A_{i-1} is to encode the output 'Adder Transfer' t^A_{i-1} into algebraically equivalent value and to use a corresponding decoder to decode the input 'Adder Transfer' into the form required by MIRBAs. k RBA-2s of a MIRBA produce k 'positive transfers' and k 'negative transfers'. Thus the value of t^A_i, t^A_{i-1} lies in the range -k to k and this can be encoded into $\lceil \log_2(2k+1) \rceil$ bits (pins). However, the corresponding decoder is too complicated. A simpler design of the encoder, MATE, and decoder, MATD, (shown dotted in Figure 4.10) results if the k positive and k negative 'Adder 'Transfers' are separately encoded into $\lceil \log_2(k+1) \rceil$ bits each. The corresponding decoder is simply a fan-out network such that the bit of weight w would fan-out to w input 'Adder Transfers' of the corresponding sign. The encoder MATE simply consists of two adder networks which form

the sum of k bits each. Each adder network requires less than k full adders arranged in approximately ($\lceil \log_3 k \rceil + \lceil \log_2 k \rceil - 2$) levels [41].

It should be noted that the decrease in total pin requirement for both t_i^A and t_{i-1}^A together from 4k to $4\lceil \log_2 (k+1) \rceil$ is obtained at the cost of introducing a new logic cell (full adder) in the radix-2^k adder design and also more delay in the generation of t_{i-1}^A.

4.2.2.5 Logic design of digit product generator (DPG) - The Digit Product Generator forms the product array of two signed radix-2^k digits. It accepts the two digits encoded in SM_r format and outputs the product array in redundant binary. The logic for the DPG consists of three parts (Figure 4.11a).

a) logic for generating the product magnitude digits,

b) logic for generating the sign of the product,

and c) logic for converting the magnitude digits to redundant binary form for input to MIRBAs. The gates required for this logic are dependent on the logic vector encodings chosen for the redundant binary digits.

For the implementation of digit algorithm 1 of microinstruction FMA (Section 3.6.2.1.1), the logic for a) and b) consists of k^2 AND gates and one exclusive-OR gate respectively. The conversion logic requires k^2 exclusive-OR gates (Equation (4.1)) or $2k^2$ AND gates (Equation (4.2)) for encoding LVE_1, k^2 AND gates (Equation (4.3)) for LVE_2 and none for the encoding LVE_3.

For the implementation of digit algorithm 2 of microinstruction FMA (Section 3.6.2.1.2), the logic for a) and b) consists of a ROM of bit capacity $2^{2k} \cdot 2k = k \cdot 2^{2k+1}$ and one exclusive-OR gate respectively.

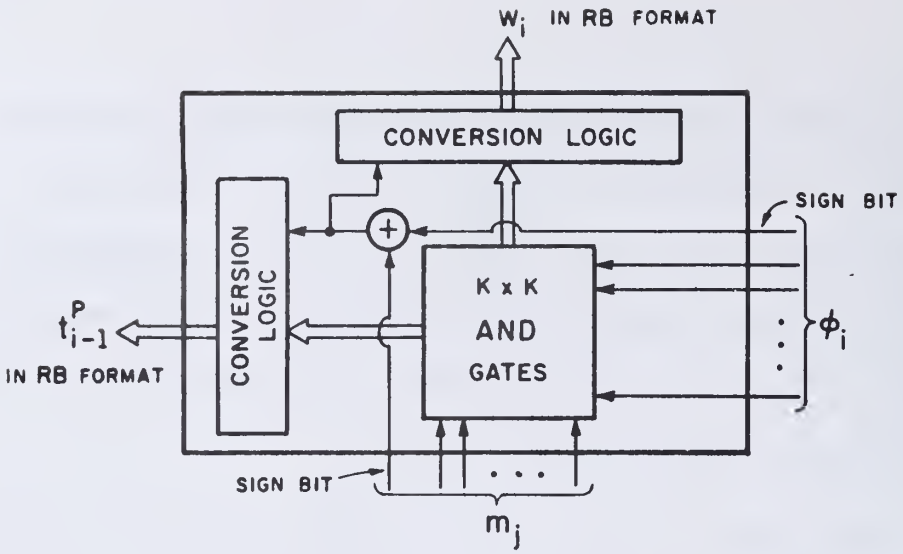


Figure 4.11a Schematic Diagram of Square Array DPG.

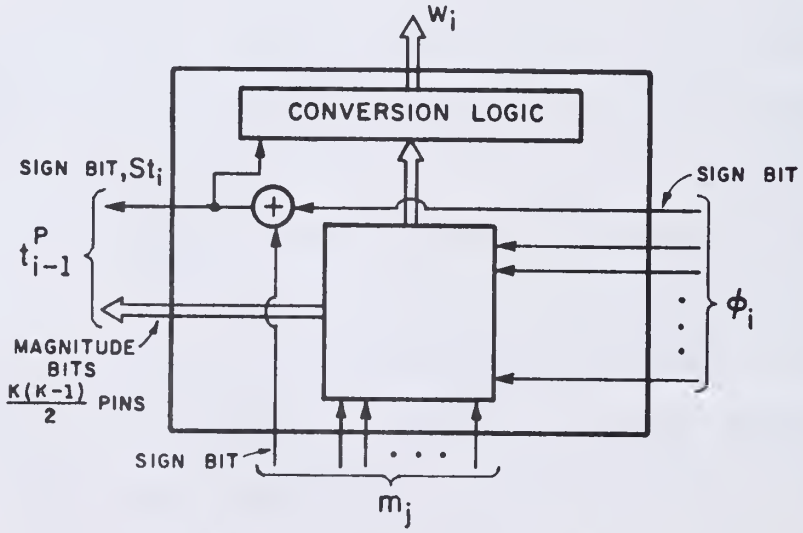


Figure 4.11b Illustration of 'Adjacent Generation' of t^P_{i-1}.

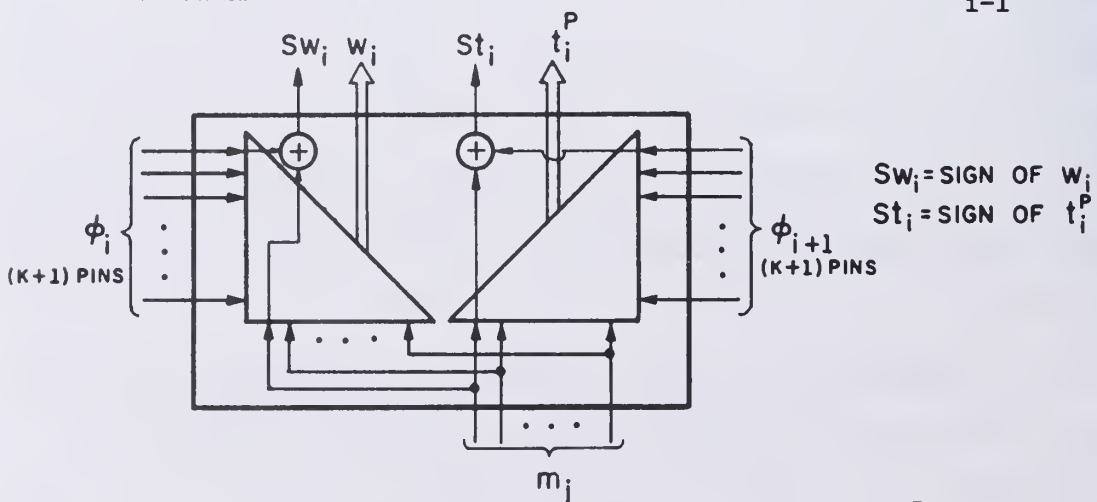


Figure 4.11c Illustration of 'Local Generation' of t^P_i.

The conversion logic, however, requires only 2k exclusive-OR gates (Equation (4.1)) or 4k AND gates (Equation (4.2)) for the encoding LVE_1, 2k AND gates (Equation (4.3)) for LVE_2 and none for the encoding LVE_3.

The pins contributed by DPG to the pin complexity of DPL are those pins which are required for the 'Collective Product Transfers' t^P_{i-1} and t^P_i. If t^P_{i-1} is generated in PE_i, then the pins needed for transmission of t^P_{i-1} to the adjacent PE_{i-1} consist of one pin for the sign of t^P_{i-1} and $(k-1)+(k-2)+\ldots+1 = k(k-1)/2$ pins for the magnitude bits, assuming that the conversion to redundant binary form is done in PE_{i-1}. We shall call this method of generating t^P_{i-1} and t^P_i as 'Adjacent Generation' (AG) of Collective Product Transfer (Figures 4.11b and 3.9).

These pins can, however, be reduced to only (k+1) from $\frac{k(k-1)}{2}$ if the CPT t^P_{i-1} (t^P_i) is generated locally in PE_{i-1} (PE_i) itself where it is needed. t^P_i (t^P_{i-1}) is a function of the multiplicand digit ϕ_{i+1} (ϕ_i) in PE_{i+1} (PE_i) and the multiplier digit m_j, the latter being the same in both PE_i and PE_{i+1} (PE_{i-1}). Thus PE_i (PE_{i-1}) needs to know only the multiplicand digit ϕ_{i+1} (ϕ_i) in PE_{i+1} (PE_i) to generate t^P_i (t^P_{i-1}), and this requires only (k+1) pins for SM_r encoded multiplicand digit ϕ_{i+1}. We shall term this method of generating CPTs as 'Local Generation' (LG) of CPT. This is shown in Figure 4.11c. Figure 4.11d shows a DPG using 'Local Generation' of t^P_i. In the LG method of generating CPTs, the logic for DPG requires one more exclusive-OR gate than for the AG method.

For the algorithm 2 of FMA, where the DPG is implemented in ROM, the pins required for t^P_{i-1} are only (k+1) -- one for sign of the product and k for magnitude bits of the product. This is shown in Figure 3.10. In

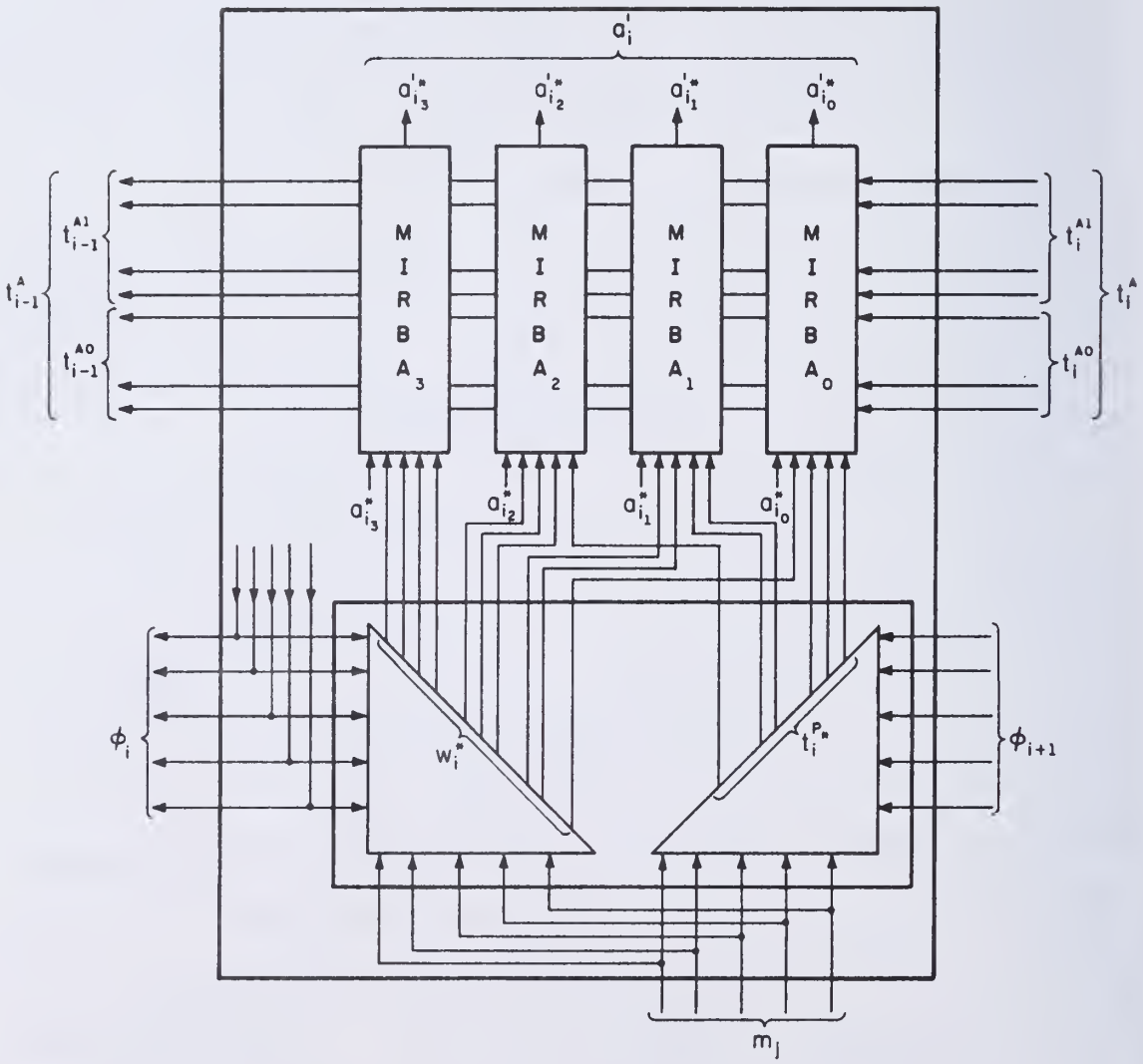


Figure 4.11d Illustration of a Combination of an MIAD and DPG using 'Local Generation' of t^P_i.

the block diagram of DPL shown in Figure 4.3, local generation of t_i^P is assumed. The register APR is used to hold the multiplicand digit ϕ_{i+1} from the adjacent PE_{i+1}.

4.2.2.6 Logic design of digit sum encoder - The Digit Sum Encoder (DSE) transforms the redundant binary sum output of the radix-2^k adder into an algebraically equivalent radix-2^k sum digit in SM_r format for either local storage in the processing element or transmission out of the PE. The DSE is an iterative logic network and involves carry propagation. Its action can be described as a two-step process.

a) determination of sign of the redundant binary sum digit and its conversion to an algebraically equivalent sum digit in 2's complement, and

b) conversion of 2's complement form of the sum digit to SM_r format. Figure 4.12a shows DSE in block diagram form. Let the input and output sum digit $\hat{x}_i$ be respectively given by (4.8 and (4.9)

$$\hat{x}_i = \sum_{j=0}^{k-1} x_{i_j}^* \cdot 2^j \qquad x_{i_j}^* \in \{\bar{1},0,1\} \quad (4.8)$$

$$= (-1)^{S_i} \cdot \sum_{j=0}^{k-1} \hat{x}_{i_j} \cdot 2^j \qquad S_{i_j}\, \hat{x}_{i_j} \in \{0,1\} \quad (4.9)$$

where $x_{i_j}^*$ is represented by a 2-tuple logic vector (χ_{i_j}, x_{i_j}) such that $\chi_{i_j}, x_{i_j} \in \{0,1\}$.

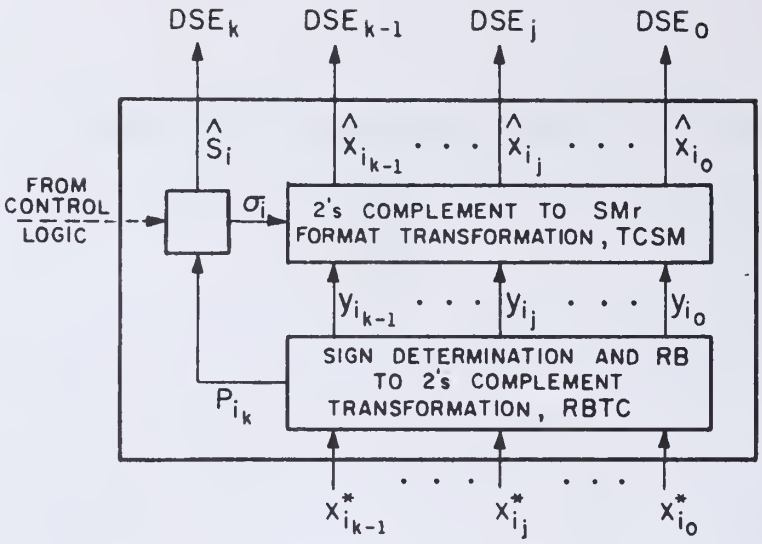


Figure 4.12a Block Diagram of Digit Sum Encoder (DSE).

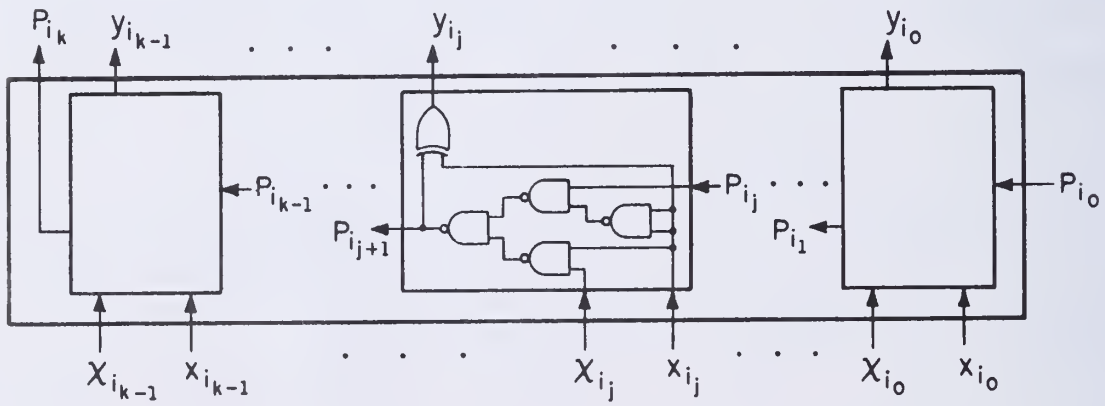


Figure 4.12b Logic Network Realization of RBTC.

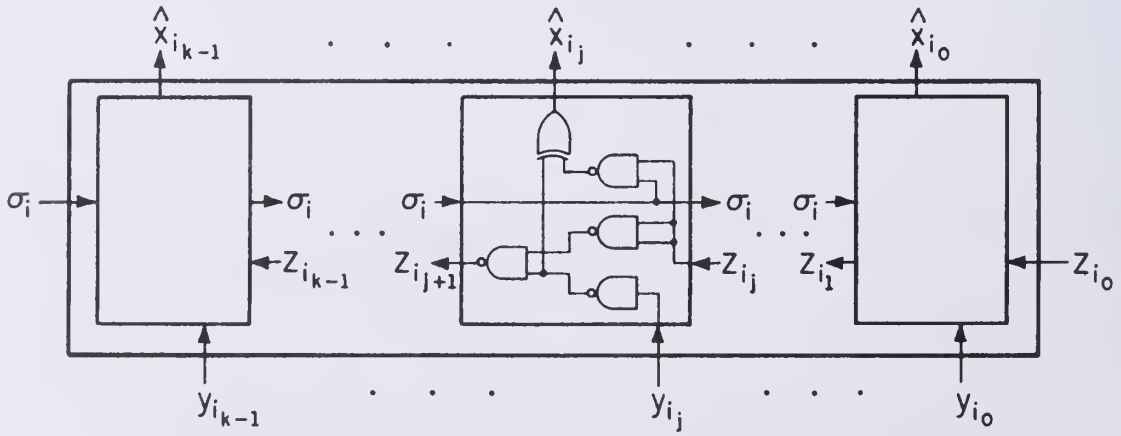


Figure 4.12c Logic Network Realization of TCSM. ($\sigma_i = P_{i_k}$)

The Redundant Binary to Two Complement (RBTC) logic (Figure 4.12b) converts input $\hat{x}_i$ to y_i such that

$$y_i = (-1)^{P_{i_k}} + \sum_{j=0}^{k-1} y_{i_j} \cdot 2^j \qquad y_{i_j} \in \{0,1\} \quad (4.10)$$

$$= \hat{x}_i$$

The logic equations of RBTC network for the three logic vector encodings of the input sum digit are given by

$\underline{LVE_3}$:

$$y_{i_j} = x_{i_j} \oplus P_{i_j} \qquad j=0,1,\ldots,k-1 \quad (4.11)$$

$$P_{i_{j+1}} = (\chi_{i_j} \wedge x_{i_j}) \vee (P_{i_j} \wedge \bar{x}_{i_j}) \quad (4.12)$$

$$P_{i_0} = 0$$

$\underline{LVE_2}$: Same as for LVE_3.

$\underline{LVE_1}$:

$$y_{i_j} = \chi_{i_j} \oplus x_{i_j} \oplus P_{i_j} \quad (4.13)$$

$$P_{i_{j+1}} = (\bar{\chi}_{i_j} \wedge x_{i_j}) \vee (P_{i_j} \wedge \overline{(\chi_{i_j} \oplus x_{i_j})}) \quad (4.14)$$

$$P_{i_0} = 0$$

The logic equations for the logic network TCSM (Figure 4.12c) that converts 2's complement form y_i to corresponding SM_r format are independent of logic vector encodings for the input sum digit. The logic equations are

$$\hat{x}_{i_j} = y_{i_j} \oplus (\sigma_i \wedge Z_{i_j}) \tag{4.15}$$

$$Z_{i_{j+1}} = Z_{i_j} \vee y_{i_j} \tag{4.16}$$

$$Z_{i_0} = 0$$

$$\sigma_i = \hat{S}_i = P_{i_k}$$

The signal Z_{i_j}, if equal to logical zero implies that the binary digits $y_{i_j}, y_{i_{j-1}},\ldots,y_{i_0}$ are all logical zero.

The Digit Sum Encoder DSE logic is also used to achieve the radix (2^k) complement and diminished radix (2^k-1) complement of the magnitude bits of ROB input to DSE via sDSE. Assuming logic vector encoding LVE_3,

$$\chi_{i_j} = 1, \qquad j = 0,1,\ldots,k-1$$

$$P_{i_0} = 0$$

$$\sigma_i = 0$$

will cause the radix complement of the magnitude bits to appear at the output of DSE, whereas

$$\chi_{i_j} = 1, \qquad j = 0,1,\ldots,k-1$$

$$P_{i_0} = 1, \ \sigma_i = 0$$

will generate the diminished radix complement of the input magnitude bits.

Similarly, $\chi_{i_j} = 0, \quad j = 0,1,\ldots,k-1$

$$P_{i_0} = 1, \ \sigma_i = 0$$

will subtract unity from the value of the input magnitude bits. These particular values of χ_{i_j}, P_{i_0}, and σ_i are made use of in the processing of microinstructions AR and NR, as described in Section 4.3.2.3.7.

However, in the case of inter-register transfer microinstructions TD and TI, the magnitude bits at the input and output are to remain unchanged; the sign bit, $\hat{S}_i$, is equal to the complement of S_{ROB} for micro-instruction TI, where S_{ROB} denotes the sign of the digit on the bus ROB.

From Figures 4.12(b) and 4.12(c), we see that RBTC and TCSM consist of k-stages each of identical logic cells. Each cell requires four 2-input NAND gates and one exclusive-OR (EX) gate. An EX-gate is equivalent to four 2-input NAND gates. Therefore the total number of gates, G_{DSE} required by DSE logic using logic vector encoding LVE_3 or LVE_2 is given by

$$G_{DSE} = 16K + C_1 \qquad (4.17)$$

where C_1 is a constant and gives the gates necessary for generation of $\hat{S}_i$ and σ_i under various control signal conditions. Use of logic vector encoding LVE_1 will raise G_{DSE} to $26K + C_1$.

In the remainder of this chapter, we shall assume only the sign-magnitude logic vector encoding LVE_3 for the redundant binary digit because

a) conversion from sign-magnitude format SM_r to RB_r format is the simplest for logic vector encoding LVE_3 for the redundant binary digit, as shown in Equation (4.4), and

b) the number of gates required for the logic implementation of the Digit Sum Encoder, DSE, is less in the case of the encoding LVE_3 than that of LVE_1 whereas the gates required for an RBA-2 are comparable for both the encodings LVE_1 and LVE_3. The encoding LVE_2 is too expensive for the logic implementation of an RBA-2 and hence of MIAD which is the major consumer of gates in the DPL.

4.2.2.7 Logic design of selector networks - Since the Adder MIAD and Digit Sum Encoder, DSE, are shared by more than one microinstruction, selector networks are needed in order to route appropriate data to the inputs of these processing logics. These selector networks also do reformatting of data, if necessary. Besides the selector networks sADR and sDSE, two more selector networks, sROB and sRIB, are necessary for transferring data out of and into the various registers of the register file. In addition, selector sTOP is used to choose the contents of output port TOP. In the following discussion, logic vector encoding LVE_3 for the redundant binary digit is assumed.

4.2.2.7.1 Logic design of adder input selector (sADR) - Selector sADR accepts inputs from two sources: 1) 'Product Array' w_i and 'Collective Product Transfer' array t_i^P along with their corresponding signs Sw_i and St_i from the output of DPG and 2) the contents R2,...,R5 of the internal registers INR2,...,INR5 of the register file. These

inputs are in sign-magnitude format, SM_r. Depending on the microinstruction, the sADR directs the appropriate data reformatted in redundant binary form to the inputs of the MIAD. For the microinstruction FMA, the outputs of DPG are routed appropriately so that the redundant binary elements of the 'product' and 'transfer' arrays are added by MIRBAs of appropriate weights. In the case of microinstruction SS, only contents R2 of register INR2 are inputed to the adder and for microinstruction MS, the contents of one or more of the four registers INR2,...,INR5 are directed to the input of the adder. The contents R1 of the accumulator register INR1 are inputed directly to the adder. The logic networks for the selector sADR for radix 16 (k = 4) are shown in Figures 4.13a and 4.13b. Figure 4.13a shows the selector for the magnitude bits and Figure 4.13b shows the generation of appropriate sign bits for the redundant binary adder inputs. SR_j (j=1,...,5) indicates the sign bit of inputs R1,...,R5.

The control signals Rj sADR (j=2,3,4,5) and SWTsADR are provided by local control logic in the PE. Since the selector networks have no memory and the data at the input of adder MIAD must be continuously available throughout the processing of microinstructions SS, MS and FMA, the selector control signals are permanently tied to the appropriate outputs of the microinstruction decoder.

For any radix-2^k, and assuming that the adder MIAD is made up of k-stages of (k+1) input MIRBAs, the gates required for magnitude and sign bits (using logic vector encoding LVE_3 for redundant binary) are respectively $3k^2$ and (3k+1). Denoting by G_{sADR} the total number of gates for selector sADR for a radix-2^k PE, we have

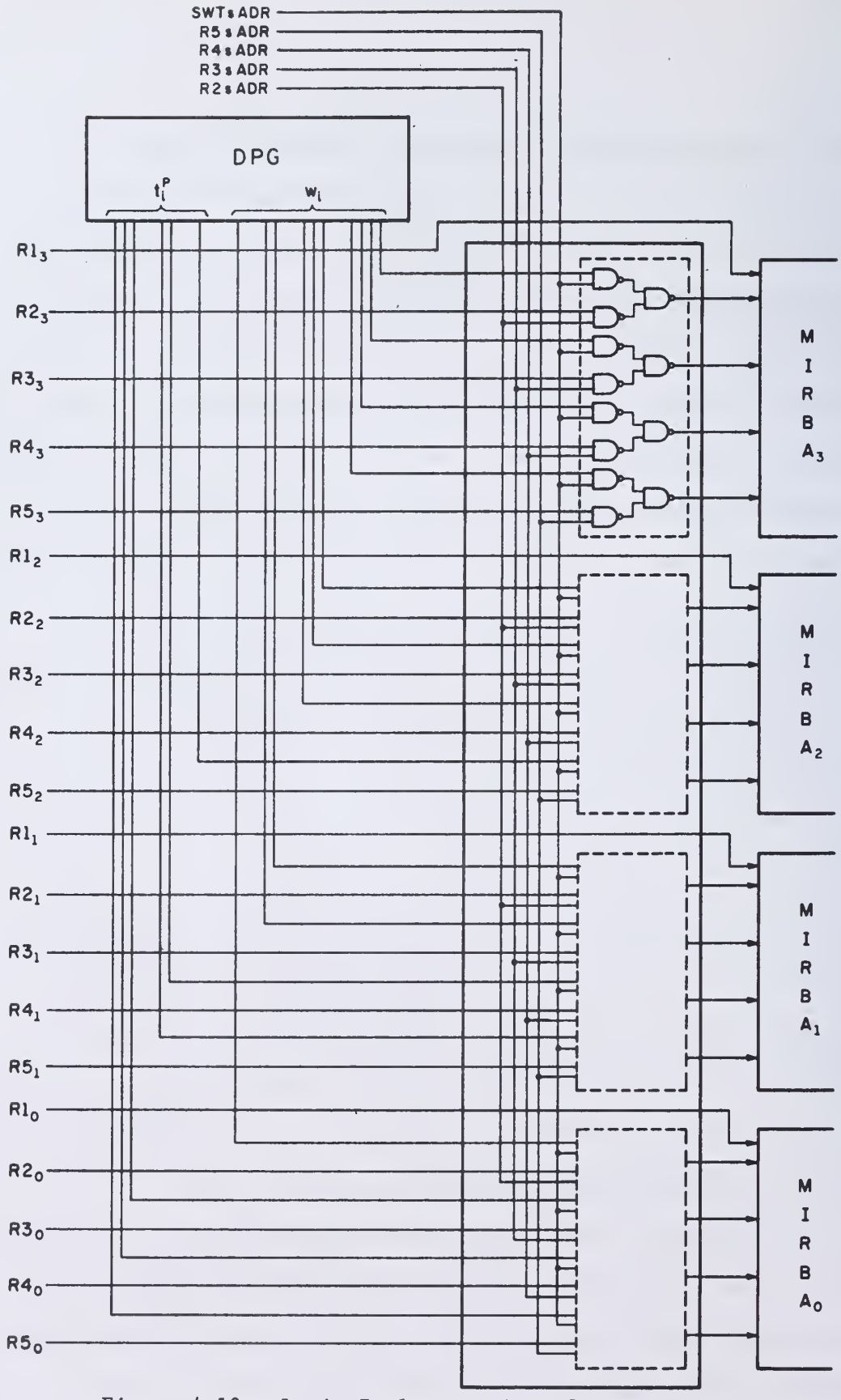


Figure 4.13a Logic Implementation of Selector sADR for Magnitude Bits.

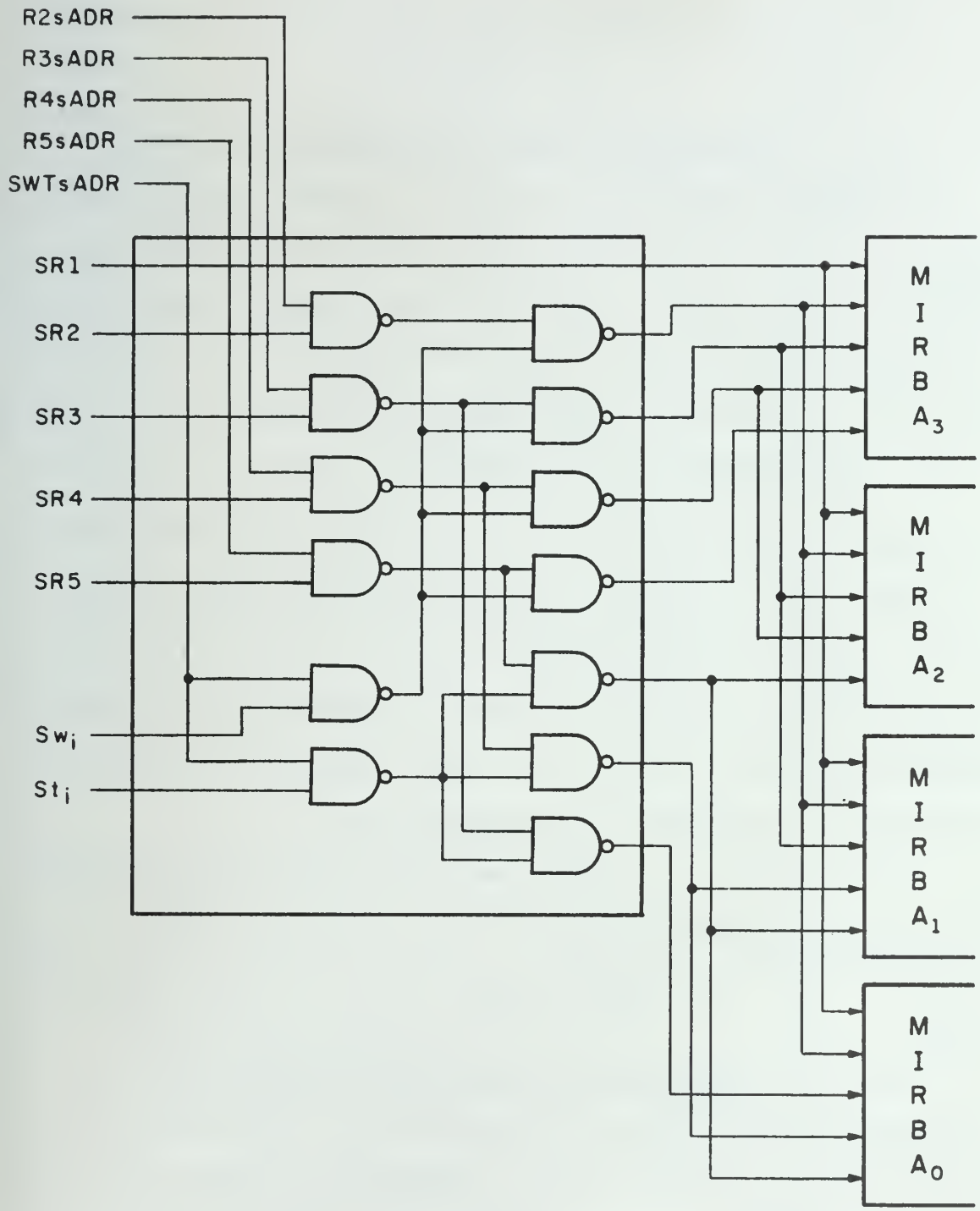


Figure 4.13b Logic Implementation of Selector sADR for Sign Bits.

$$G_{sADR} = 3k^2 + (3k+1)$$
$$= 3k^2 + 3k + 1 \qquad (4.18)$$

4.2.2.7.2 Logic design of digit sum encoder selector (sDSE) -

The selector sDSE (shown in Figure 4.14) accepts inputs from two sources--the two outputs a^*SS_j and a^*MF_j (j=0,1,...,k-1) of the adder MIAD and the ROB_j (j=0,1,...,k-1). The control signals ASSsDSE, AMFsDSE and ROBsDSE, respectively select the MIAD outputs a^*SS_j and a^*MF_j corresponding to microinstructions SS, FMA and MS and the bus ROB. The control signal SCHI appropriately sets the sign bit χ_{i_j} (j=0,1,...,k-1) of the redundant binary input $\chi^*_{i_j}$ of the DSE to achieve radix complements or diminished radix complement or direct transfer of the magnitude bits of ROB. This was explained earlier in Section 4.2.2.6. Figure 4.14 shows the logic implementation of sDSE for radix-16. Since the selector sDSE has no memory, the appropriate control signals must be held active throughout the processing of the microinstruction.

For any radix-2^k, the total number of NAND gates, G_{sDSE}, required for the logic of sDSE is given by

$$G_{sDSE} = 7k. \qquad (4.19)$$

4.2.2.7.3 Logic design of selectors sRIB, sROB and sTOP - The selector sRIB is a three input multiplexer and has three input sources--the DSE, APR and the digit field of microinstruction register MIR in the control logic of PE. The selector is one digit wide and the NAND gates required for the logic implementation of this selector shown in Figure 4.15 is given by G_{sRIB} where

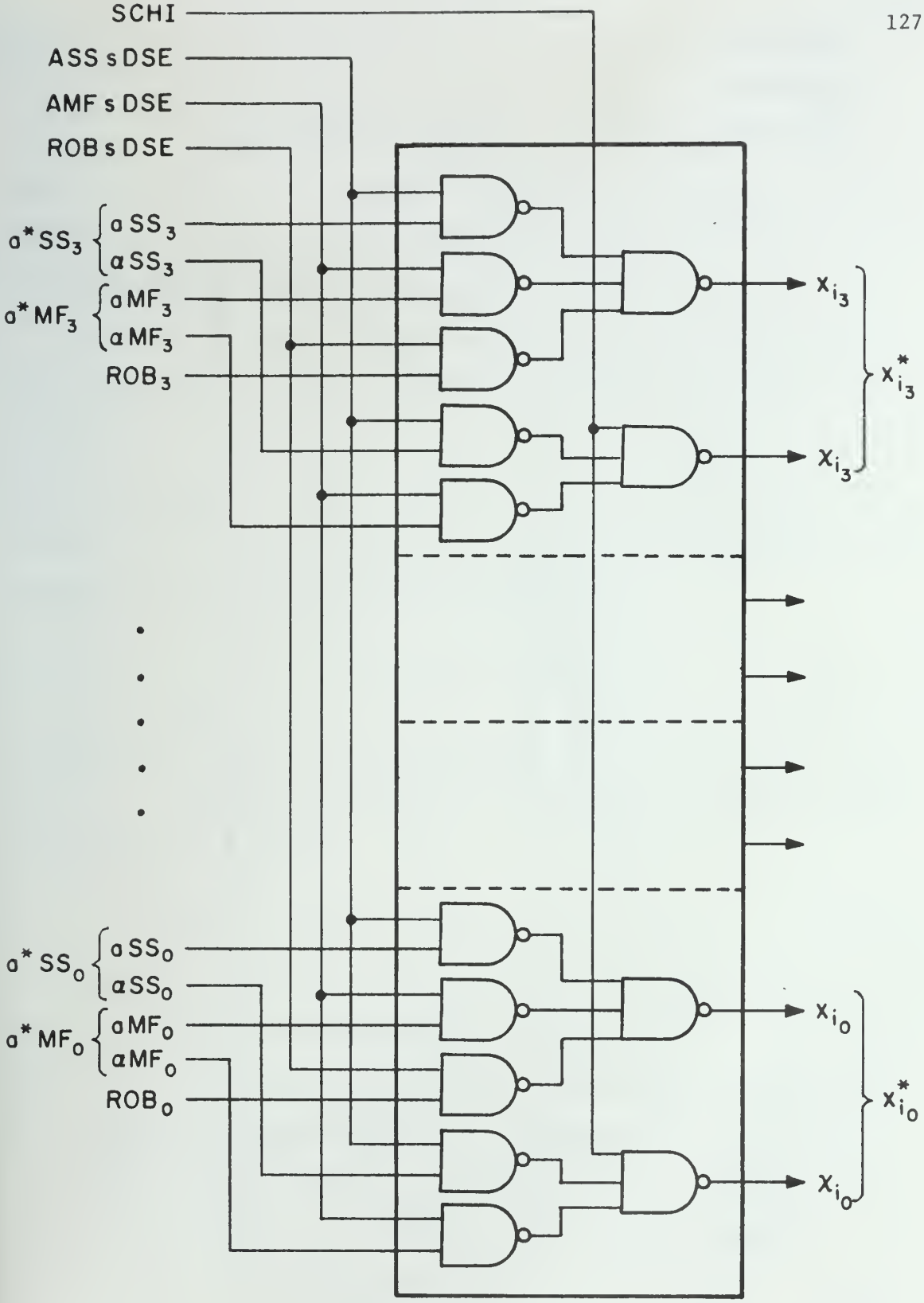


Figure 4.14 Logic Implementation of Selector sDSE.

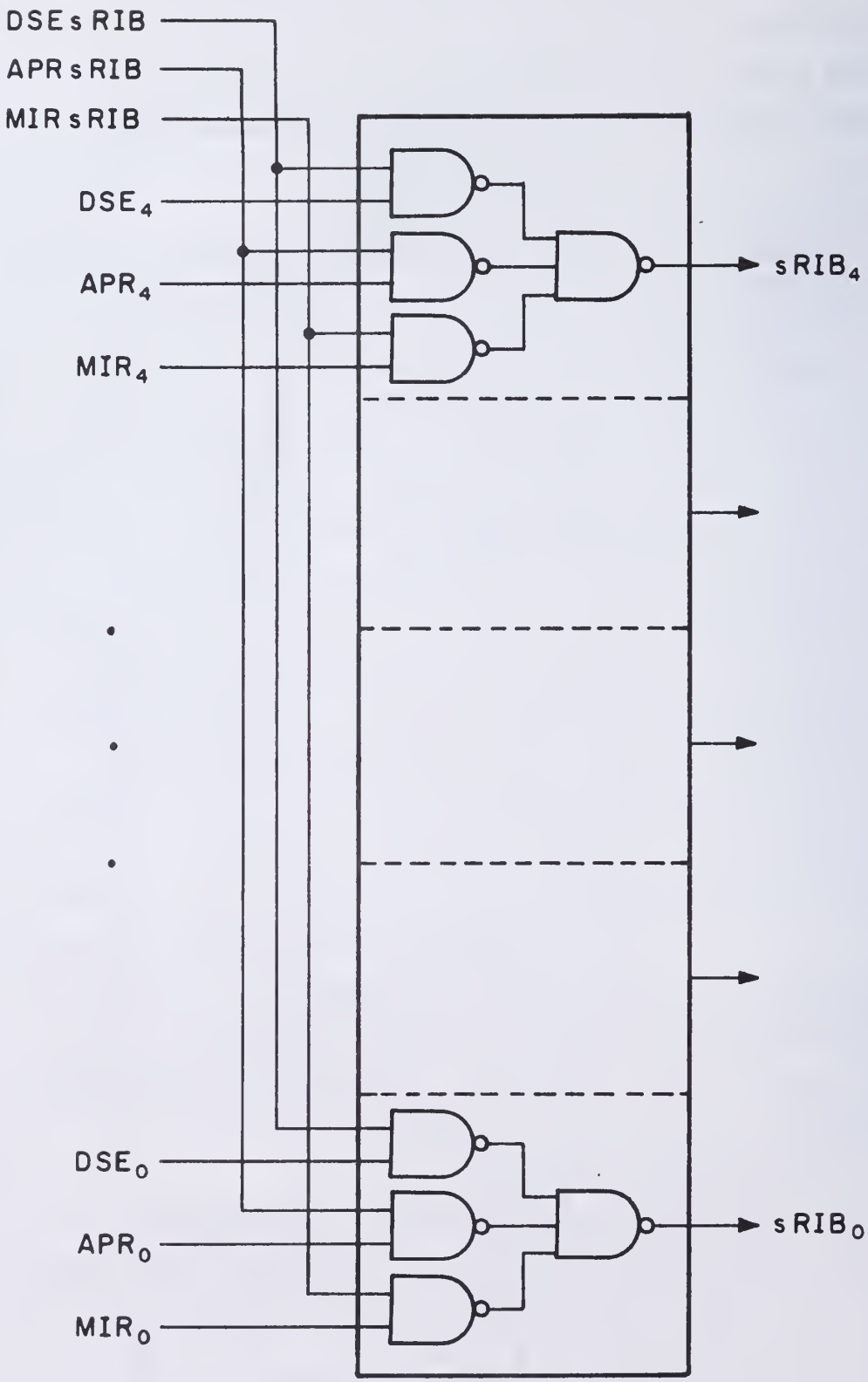


Figure 4.15 Logic Implementation of Selector sRIB.

$$G_{sRIB} = 4(k+1). \qquad (4.20)$$

The control signals DSEsRIB, APRsRIB and MIRsRIB, respectively select the sources DSE, APR and MIR. The path from MIR to sRIB is made use of in the processing of microinstructions RS and LPM.

The width of selector sTOP (Figure 4.16) is equal to the width of output port TOP_i. The width of TOP_i is determined by the number (=k+1) of bits required for the address space of PEM_i plus one more for the Read/Write function of PEM_i and the bits, $P^A_{t_{i-1}}$ required for the 'Adder Transfer' t^A_{i-1} which is dependent on the method of encoding used for t^A_{i-1}. Assuming the width of TOP_i to be b, b is given by

$$b = Max(k+2, P^A_{t_{i-1}}).$$

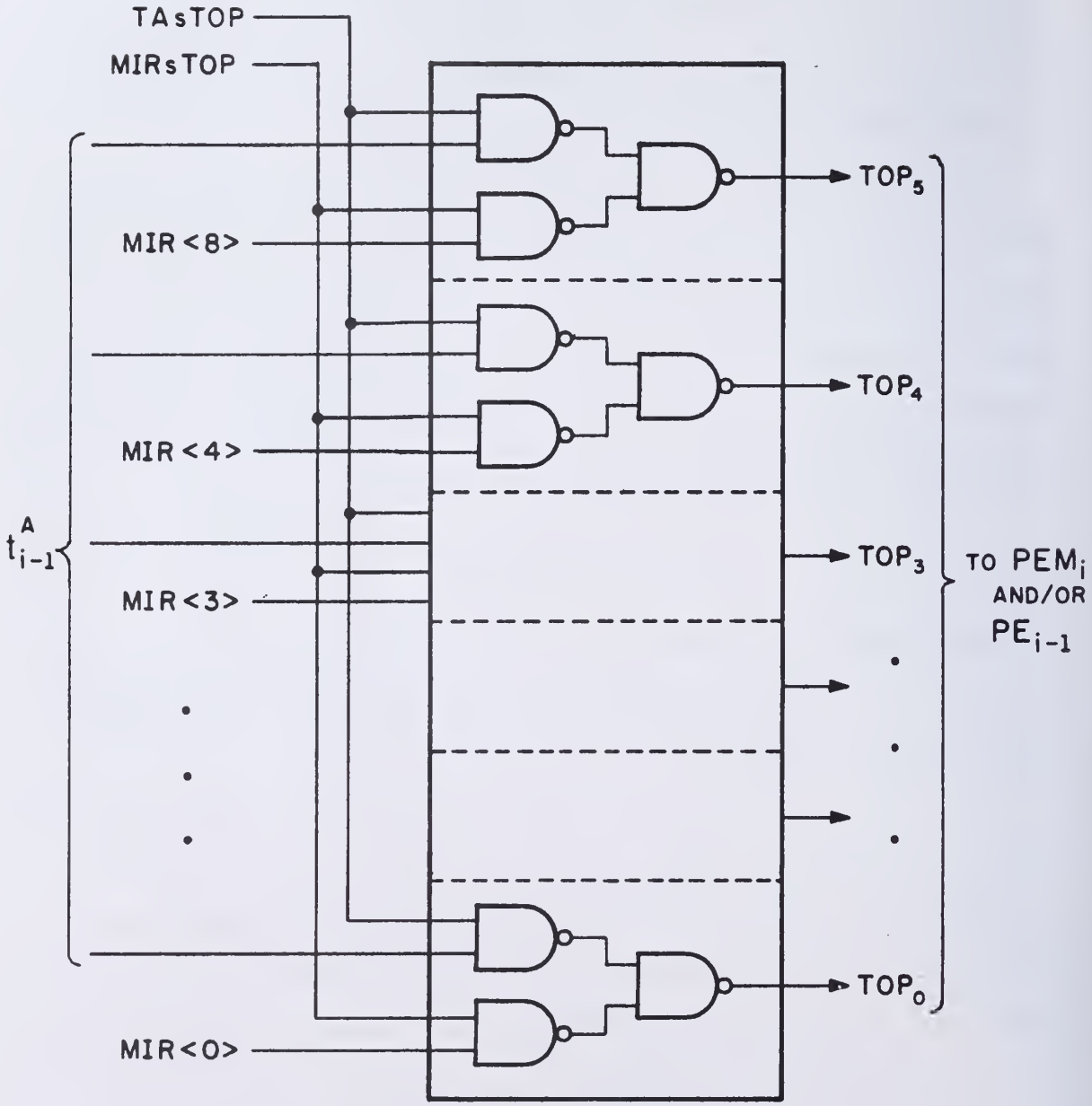
For MIADs using encoders and/or redundancy ratio $\delta \leq 2/3$, k+2 is greater than $P^A_{t_{i-1}}$. Therefore the gates, G_{sTOP} required for the logic implementation of selector sTOP is

$$G_{sTOP} = 3(k+2). \qquad (4.21)$$

The selector sROB selects the contents of one of the registers of the register file on to the register file output bus ROB. The gates required for this network are dependent on the number of registers in the register file and the bit width of the registers. For radix-2^k, the register width is (k+1) bits and assuming (k+1) registers in the register file, the total gates required are

$$G_{sROB} = (k+1)(k+2). \qquad (4.22)$$

Figure 4.17 shows logic implementation of sROB for radix-2^k (k=4).



Note: MIR<4:0> are PEM address bits
MIR<8> is Read/Write bit

Figure 4.16 Logic Implementation of Selector sTOP.

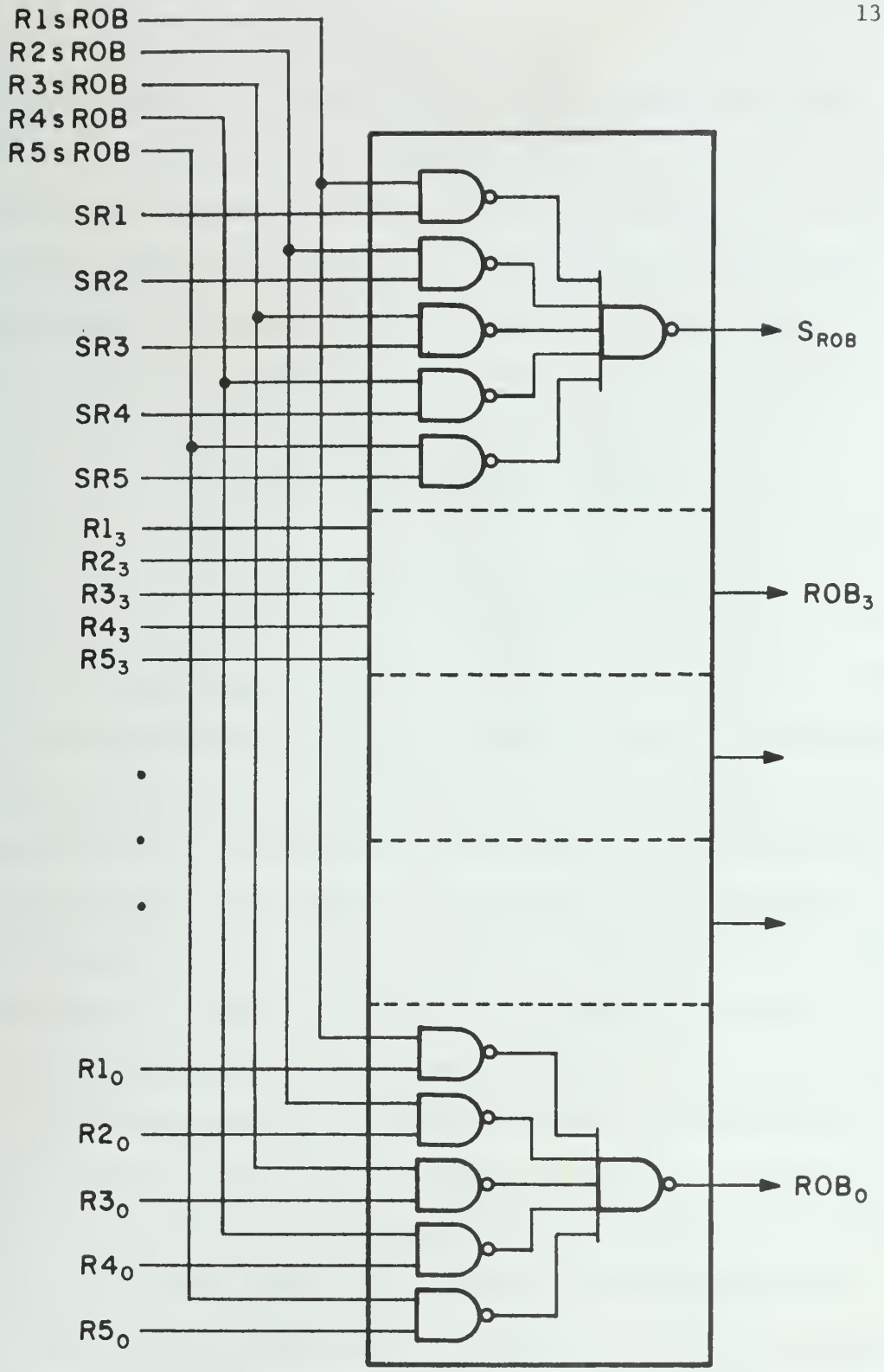


Figure 4.17 Logic Implementation of Selector sROB.

Note that these selectors have no memory. The control signals would have to remain active throughout the processing of a micro-instruction. Although the selectors are shown to have separate control signals, fewer control signals with a local decoder would suffice. But the separate control signals are shown for the ease of exposition because the separate names of the signals help to identify the sources easily.

4.2.2.7.4 Storage buffer registers of DPL - In addition to the combinational logic for processing and the selector networks for proper data routing, the DPL has three buffer registers GIR, APR and IB. GIR and APR are used to hold the G-information from the adjacent PE. The register GIR holds the 'Adder Transfer' t_i^A from the adjacent PE_{i+1} and the register APR stores the multiplicand digit ϕ_{i+1} for the local generation of 'Product Transfer' t_i^P in the DPG. The width of APR is (k+1) bits for radix-2^k and the width of GIR depends upon the bit requirements of t_i^A. At the maximum, it is equal to 2k if no encoder MATE is used in the design of adder MIAD. However, if either the number of inputs to the MIRBAs is reduced by changing the redundancy of the multiplier digit or by any other means, then the bit width of GIR would be correspondingly reduced. Assuming that the 'encoders' and 'decoders' MATE and MATD are not used for the t_i^A and t_{i-1}^A, and the inputs to each MIRBA is k' for radix-2^k adder, then the bit width of GIR is 2(k'-1).

The register APR is also used in the processing of left shift microinstruction LS and is used to hold the shifted digit from the right neighbor PE_{i+1} temporarily before being stored in the register file via the selector sRIB.

Since the outputs of internal registers of the register file are directly and permanently connected to the input of combinational processing logic, it is necessary to provide a buffer register, IBR at the output of selector sRIB. The output of the selector sRIB is gated into the register IBR and thus isolates the input bus of the registers from any changes which might occur due to feedback through the combinational logic when the contents of the buffer register IBR (i.e., the result digit) are transferred to the appropriate register in the register file.

The bit-width of the register IBR is (k+1) for a radix-2^k digit.

4.3 Design of PE Control

The processing of a microinstruction in the PE requires the activation of the various data paths and the conditioning of combinational transformation logic of the DPL, in a certain temporal order depending on the nature of the microinstruction. These time ordered activation control signals are generated by the PE Control Logic (PCL) which is locally resident in the PE.

Another function of the local PE control is to coordinate the actions of the PEs not only to obtain 'G' information from adjacent PEs for the processing of the microinstruction but also to receive and transmit the microinstructions from and to the adjacent PEs. The latter is necessary to process a 'machine instruction'. Each PE executes the same sequence of microinstructions which is issued by MCU depending on the 'machine instruction' to be processed by the Arithmetic Unit and the specific operand values. After executing microinstruction

j-1, a typical PE, PE_i say, must determine the value of ${}_jG_i$ and inform the PE_{i-1} of its availability. ${}_jG_i$ is needed by PE_{i-1} to execute microinstruction j. PE_i also passes the jth microinstruction and modifier to PE_{i+1} so that PE_{i+1} will determine ${}_jG_{i+1}$, in cooperation with PE_{i+2}, if necessary. When PE_i receives ${}_jG_{i+1}$ it performs the microinstruction j and begins the procedure for microinstruction j+1.

The control strategy for implementing the coordination of the various PEs can be either synchronous or asynchronous. In the former case, all the PEs act in synchronism with some central clock whereas in the asynchronous case, all the activities are controlled by request-response signals. In this paper, asynchronous control with request-response signals is chosen because of the following advantages:

a) It avoids the clock-skew problems when a large number of PEs are concatenated together for high precision of arithmetic.

b) Due to the pipeline nature of processing, different PEs at any instant are executing different microinstructions which take different times to execute. The request-response strategy will provide overall better average speed of processing.

c) The asynchronous control is compatible with the 'localized' nature of processing and an autonomous and modular arithmetic element.

However, it does have the disadvantage of increasing the number of pins required for the PCL.

4.3.1 Logical organization of PE control - The PE control is organized as a set of six interacting subcontrols some of which are active concurrently while others are activated in sequence, depending on the nature of the control algorithm for the microinstruction. Concurrently interacting controls allow an average speed up in the processing of microinstructions by allowing independent operations to take place in parallel.

Figure 4.18 shows the various subcontrols and their interaction. The division of PE control into subcontrols is based on a functional grouping of the various steps in the control flow. The various subcontrols are R-control, T-control, G-control, E-control, F-control and DM-control.

The Decode and Main or DM-control is the main control which supervises and coordinates the actions of other subcontrols. It handles the decoding of the microinstruction, sets up the necessary data paths in DPL, and then chooses the proper subcontrols and their temporal order for the execution of the control algorithm of the microinstruction. (In a crude software analogy, the DM-control can be considered as the Main procedure and other subcontrols which are invoked by DM-control as subroutines.)

The Receive or R-control and the Transmit or T-control are the primary controls for the coordination of PEs. R-control is concerned with accepting the microinstruction from the left neighbor PE_{i-1} and acknowledging the receipt of the microinstruction (OP-code μ_j and the modifier field ${}_jF_i$). The T-control transmits the received microinstruction with the same or a new modified F-field ${}_jF_{i+1}$, depending on the nature of the microinstruction, to the PE_{i+1}.

The G-control and E-control together can be considered as constituting the main processing controls for the microinstruction. The G-control generates the G-information for the left neighbor PE_{i-1} and accepts the G-information from the right neighbor PE_{i+1}. The Execute or E-control activates the necessary control signals to the combinational logic to calculate and gate the result digit in appropriate internal register of the register file. In addition to this, the status of the digit in the accumulator register is set. The status checking involves determining the sign and magnitude of the digit. If the accumulator digit is zero, the sign of the digit is considered to be unknown.

The F-control is used when a new value, different from that received, of the modifier field has to be sent to the right neighbor PE_{i+1}. It is made use of in right shift microinstruction RS.

4.3.1.1 Global description of interaction of subcontrols -

Figure 4.18 shows the interaction of the various subcontrols. It should be noted that Figure 4.18 does not show the hierarchical order in which the various subcontrols are invoked by DM-control but only shows a gross overview of the interaction. The specific temporal order of the various subcontrols in the control sequence of any microinstruction is discussed later in Section 4.3.2.3.3.

The control sequence for every microinstruction begins in the R-control. The R-control, on receiving a go-ahead signal from DM-control to accept another microinstruction from the left neighbor PE_{i-1}, accepts the microinstruction and acknowledges back the receipt of the microinstruction. It also invokes the DM-control. The DM-control decodes

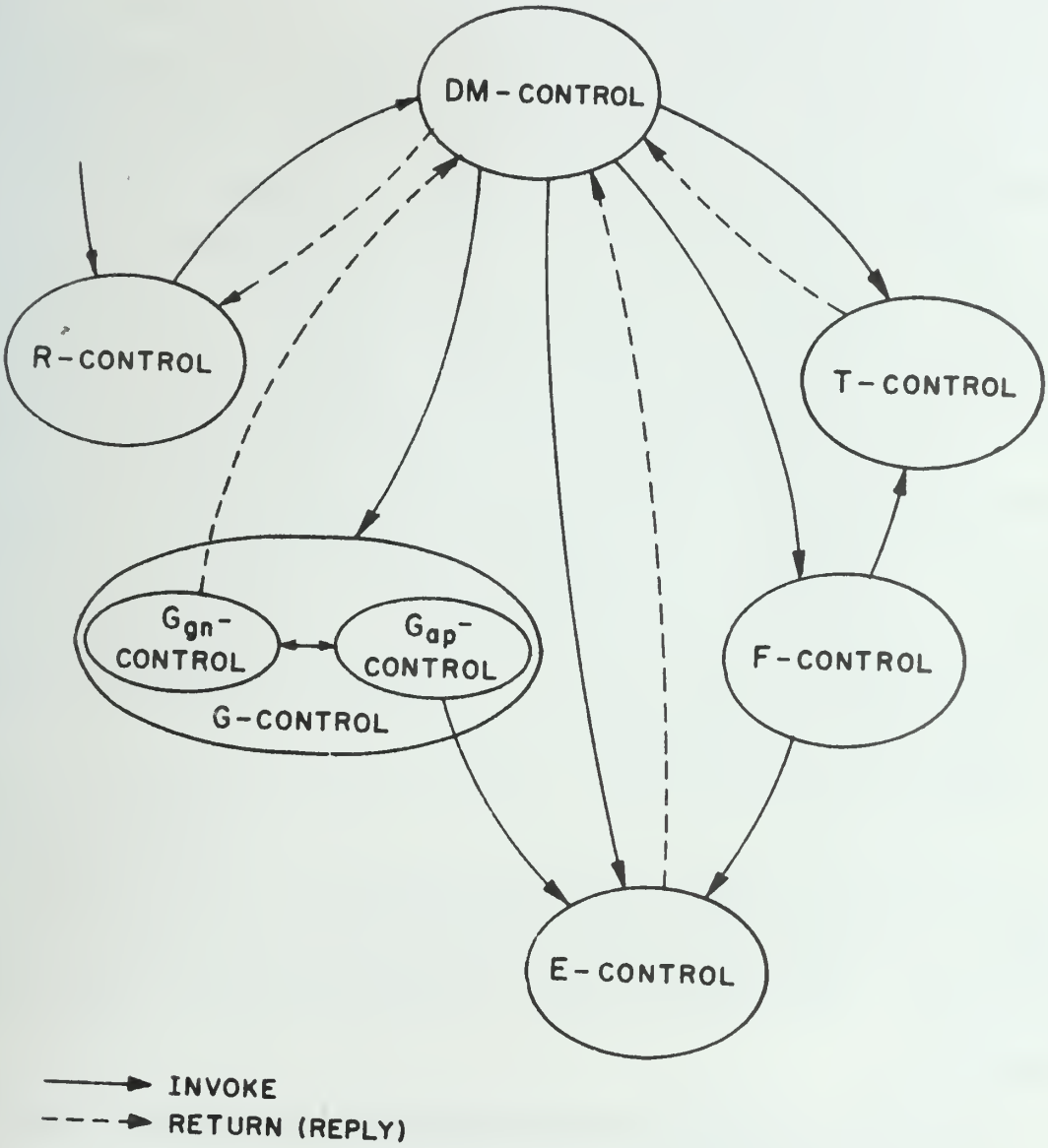


Figure 4.18 Logic Organization of PE Control Signal Generator.

the microinstruction, sets up the data paths in the DPL and invokes one or more of F, G, T, and E controls depending on the microinstruction type. The F-control makes the changes in the modifier field of the microinstruction and calls on the T-control to transmit the modified microinstruction to PE_{i+1}. F-control is invoked only for right shift microinstruction RS. If the processing of microinstruction requires G-information, the G-control and T-control are invoked in parallel. The G-control can be conceptually considered as comprising of two subcontrols: G_{gn}-control which generates G-information for the microinstruction executing in adjacent PE_{i-1}, and G_{ap}-control which accepts G-information from the right neighbor PE_{i+1}. (In the case where G-information depends logically on two or more right neighboring PEs (e.g., microinstructions FMA, AR), the subcontrols G_{gn}-control and G_{ap}-control interact with each other.) After the necessary G-information for the execution of the microinstruction has been obtained, the G_{ap}-control branches to E-control for the execution of the microinstruction.

In those cases when G-control is not invoked by DM-control because no G-information is needed from adjacent neighbors (e.g., microinstructions TD, TI, LDC), the DM-control directly calls upon E and T controls in parallel. The T-control transfers the microinstruction to the right neighbor PE_{i+1}.

As the various invoked subcontrols finish their sequence operations, they report back to the DM-control. When all the invoked subcontrols are finished, the DM-control replies back to the R-control which was suspended earlier from accepting any more microinstruction. The R-control now is again ready to accept another microinstruction and the control sequence begins again.

4.3.2 Logic design of PE control

4.3.2.1 Block diagram description of PE control logic (PCL) -

Figure 4.19 shows the major components of the PCL in block diagram form. It consists of a microinstruction register MIR, the selector network, sMIR, the 'Zero magnitude and Sign Detector', ZSD, and the timing control signal generator, TCS. The register MIR is 11 bits wide and is used to hold the microinstruction, received on microinstruction input port, MIP_i, from adjacent PE_{i-1}, during processing by PE_i. The selector sMIR is a two way multiplexer which chooses either MIP_i or ROB from the DPL as the appropriate source of data for the bits <4:0> of MIR. The ZSD is a combinational logic block which monitors the sign and magnitude bits of the accumulator register INR1. It sets flip-flop Z_{i+1} to logical state '1' if the magnitude of the accumulator in PE_i is zero. Flip-flop S_i is set to the state of the sign bit SR1 of accumulator register. The TCS generates the timing signals for the activation of data paths and processing logic in DPL and for the coordination of the adjacent PEs.

The generation of the appropriate control signals and their temporal order depends on the microinstruction--its digit algorithm and the data flow structure of DPL.

4.3.2.2 Design and description of microinstruction formats - The major consideration in the design of the various microinstruction formats are:

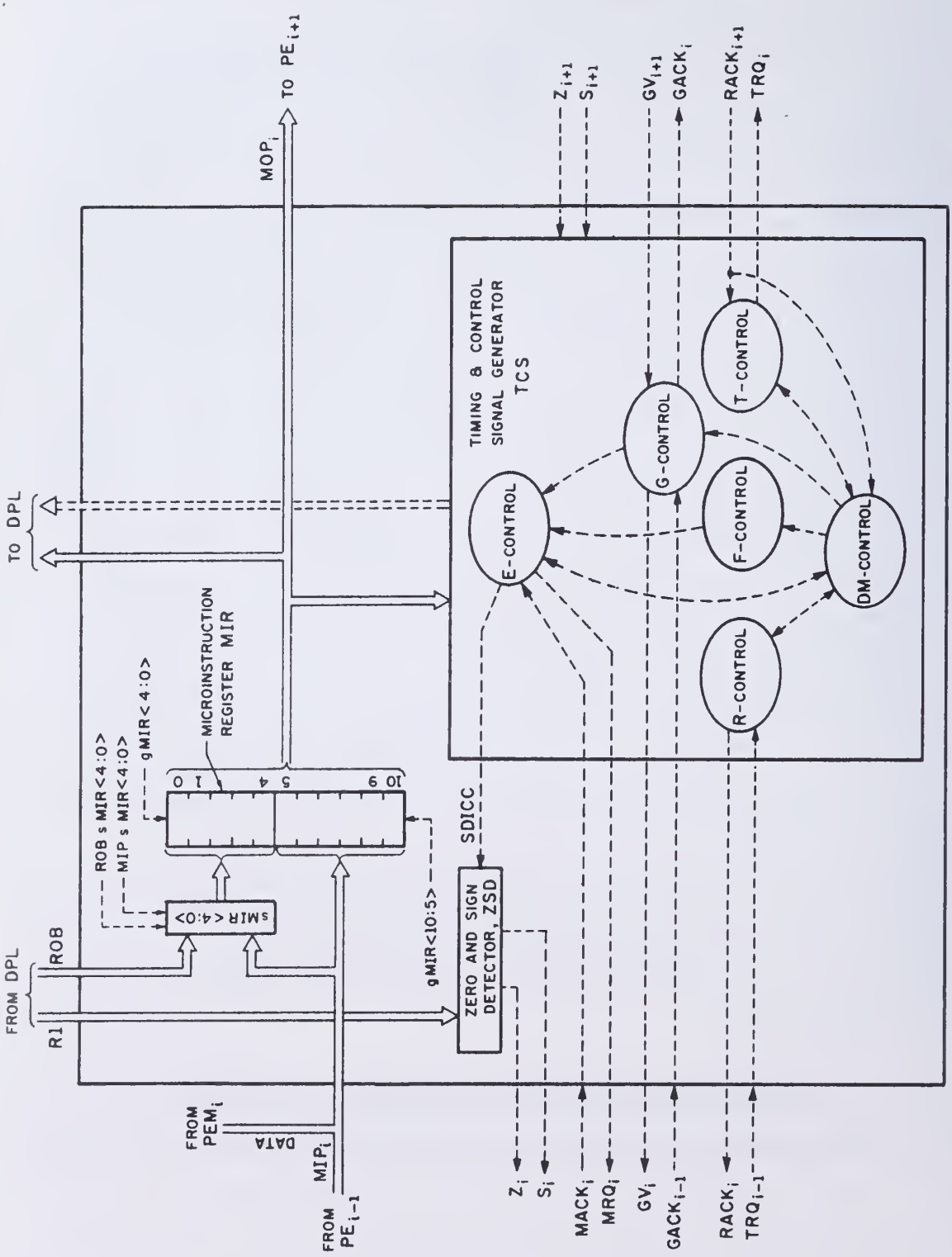


Figure 4.19 Block Diagram of PE Control Logic.

The major consideration in the design of the various microinstruction formats are:

a) the bit width of the microinstruction should be as small as possible so that the pins required for the input port MIP_i be least, and

b) the microinstructions should be powerful so that they take full advantage of the data flow structure of the DPL and facilitate the microprogramming of the 'machine instruction'.

These two aims are conflicting in nature because b) requires a large instruction width. A compromise was achieved by using varying number of bits for the OP-code of the microinstruction.

Basically, each microinstruction has an OP-code field μ_j and a modifier field, ${}_jF_i$ as was discussed in Section 2.6. The basic OP-code field is 3 bits long and the modifier field depends on the bit width (radix of arithmetic processing) of the PE and the number of addressable registers in the register file. The modifier field ${}_jF_i$ is further divided into two subfields--one field carries the address of the register in the register file of the PE and the other field carries either a digit, or the address of the PEM location in local operand mantissa memory. For some microinstructions, these fields are used for other purposes.

The Figure 4.20 shows the specific OP-code bit assignment and the formats for various microinstructions. In this figure, it is assumed that the bit width of the PE is 5 bits (that is, radix is 16) and that there are 5 (=k+1) registers in the register file of the DPL. The microinstructions LPM, SPM, RS, LS and LDC have three bit OP-codes whereas microinstructions TD and TI have four bit OP-codes. The OP-codes for

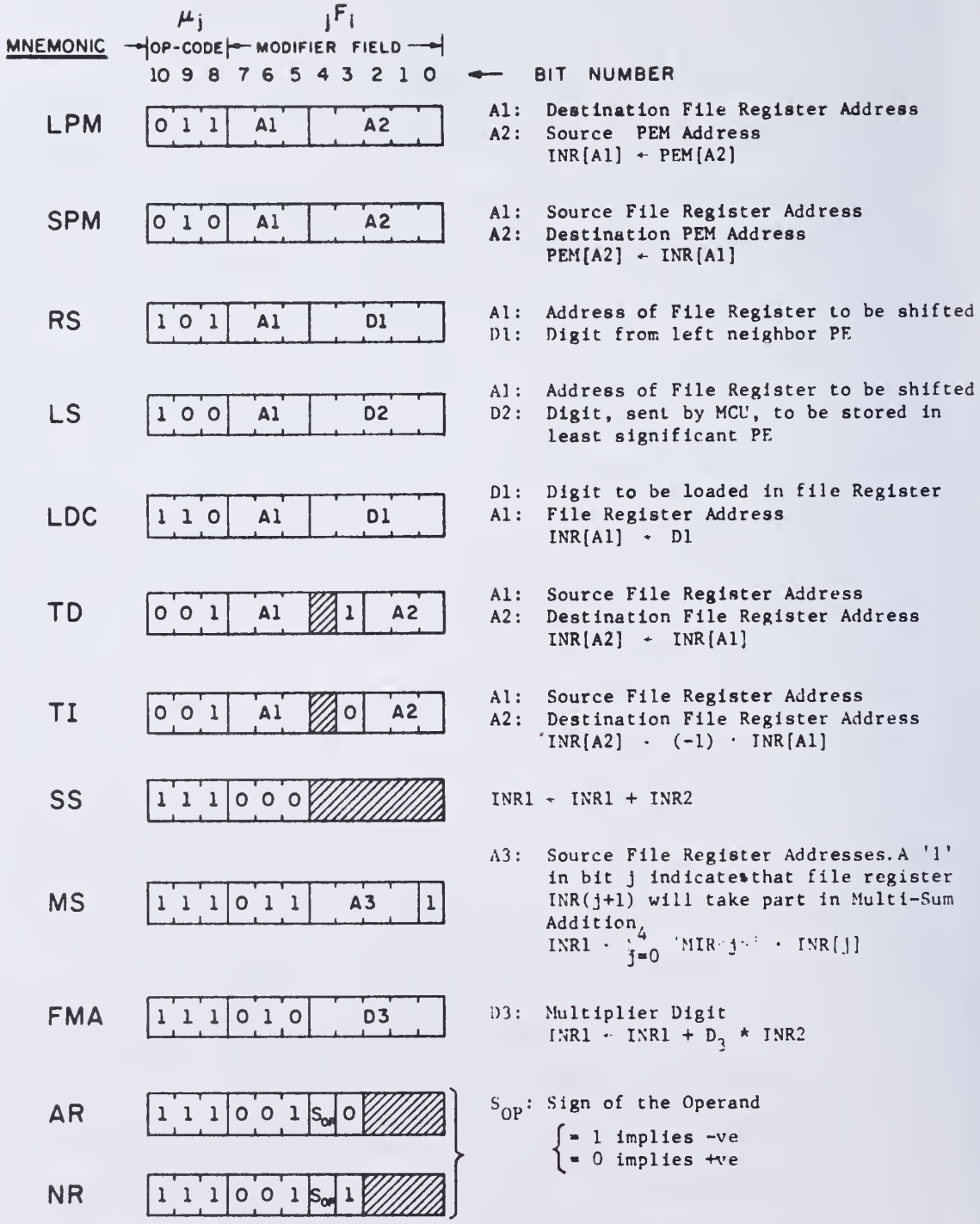


Figure 4.20 Microinstruction Codes and Formats.

microinstructions SS, MS and FMA are six bits long whereas for AR and NR, they are seven bits long. The varying length of the OP-code allows a basic three bit field for OP-codes, otherwise a straightforward coding of 12 microinstructions would have required four bit OP-codes.

It should be noted that the use of a more restricted set of microinstructions could have reduced the bit width of the microinstructions at the cost of less flexibility in microprogramming capability.

In general, for a radix-2^k arithmetic structure, the bit width of a PE digit is (k+1), and assuming the register file to consist of (k+1) registers, the bits required for a microinstruction are given by

$$\begin{aligned} I_b &= \text{Instruction width in bits} \\ &= 3 + \lceil \log_2(k+1) \rceil + (k+1) \\ &= k + \lceil \log_2(k+1) \rceil + 4. \end{aligned} \tag{4.23}$$

A description of the various microinstructions was given earlier in Section 2.6. The function of each microinstruction, briefly, is again given below.

The memory access microinstructions LPM and SPM are respectively used to fetch data from and store data to the processing element memory PEM associated with the PE. The microinstruction field A2 gives the location in PEM and field A1 identifies the register in the register file.

In the shift microinstructions RS and LS, A1 identifies the register to be shifted. The field D1 carries the digit from the register in the left adjacent PE and D2 identifies the digit which must be

loaded in the register of the least significant PE. This facility is made use of in multiplication where the digit shifted out of the most singificant PE, during left shift of partial products, has to be saved in the least significant digital position of the multiplier operand register.

The field A1 in microinstruction LDC identifies the register to be loaded with the digit given in field D1.

In microinstructions TD and TI, the A1 and A2 respectively identify the source and destination registers in the register file. Note that A2 can be equal to A1 in microinstruction TI, whereas such a condition in TD is meaningless.

In the case of arithmetic instruction SS, no special registers are identified because this microinstruction always causes the contents of accumulator register INR1 and operand register INR2 to be added with the result going to the accumulator register.

For microinstruction MS, field A3 identifies the various registers of the register file whose contents would be added by microinstruction MS. Note that the address in A3 is not encoded but rather each bit of A3 identifies a register. A bit value of '1' in A3 indicates that the corresponding file register would take part as the source of the operand. The '1' in the least significant position of A3 indicates that accumulator register INR1 would always be one of the source registers in the MS instruction. The result of addition always goes to the accumulator register INR1.

The D3 field in microinstruction FMA identifies the multiplier digit for the formation of the partial product.

D3 field in microinstruction FMA identifies the multiplier digit for the formation of the partial product.

The microinstruction bit 4 carries the sign of the operand, S_{OP}, which is nothing but the sign of the most significant nonzero digit in the accumulator. This sign is first determined by the MCU by a sequence of left shift microinstructions and testing the status indicators Z_1 and S_1 of the most significant PE_1. The proper value of S_{OP}, that is, bit 4, is set by MCU before issuing the microinstruction.

4.3.2.3 Description of subcontrols by control sequence charts -

The subcontrols are multi-output finite state machines which produce control signals in proper temporal order for the execution of various microoperations during the processing of a microinstruction. These control signals condition the combinational processing logic to perform elementary microoperations like opening or closing of a register gate, setting of selector networks to certain states or the setting of a control status memory element. In addition, some of the control signals act as interface request-response signals for the coordination of various PEs or to access the local memory (PEM) module.

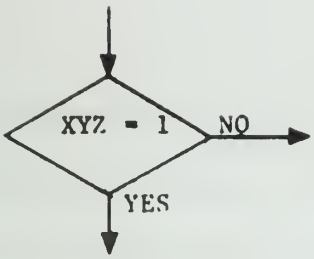
The operation of the finite state machine can be described by a control sequence chart (CSC) which is a flowchart like description of a control sequence. A control sequence is an instance of the execution of a subcontrol. The control sequence chart shows the various control signals and their temporal order generated during the execution of the subcontrol.

4.3.2.3.1 Control sequence chart conventions - A control sequence chart (CSC) consists of a set of rectangular, diamond and pentagonal shaped boxes and entry and exit symbols connected together in a two-dimensional pattern with straight directed lines. The arrows on the lines indicate the direction of the control flow in the sequence. The various symbols used in the CSC are shown in Figure 4.21.

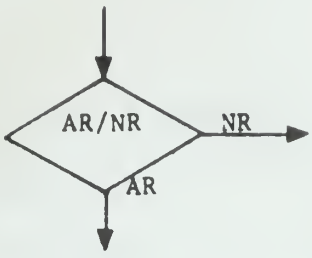
The diamond shaped symbol (Figure 4.21a and 4.21b) represents the decision element with single entry and two exit points. The exit points are labeled yes/no (Figure 4.21a) which indicate the truth/falsehood of the statement written inside the box, or the exit points are labeled with the actual name of the option (Figure 4.21b) that is valid on that exit point.

The rectangular box of Figure 4.21c represents a control step. A control step is a set of microoperations (indicated by control signals) enclosed in the rectangular box. The time ordering of the micro-operations within a control step is not important and they are, in general, all activated in parallel. The rectangular boxes of Figures 4.21d and 4.21e represents the invoking of another subcontrol whose name is written inside the box. However, in the case of Figure 4.21d, the exit from the subcontrol returns the control flow to the point where it was invoked (like a subroutine call in software) whereas the control flow at the end of execution of the subcontrol indicated in Figure 4.21e branches to the next point in the control sequence chart.

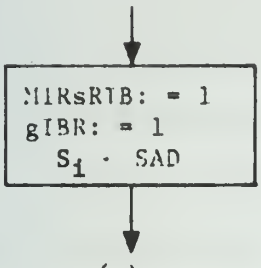
The pentagonal boxes of Figures 4.21f and 4.21g respectively represent the 'FORK' and 'JOIN' symbols. The 'FORK' symbol indicates that



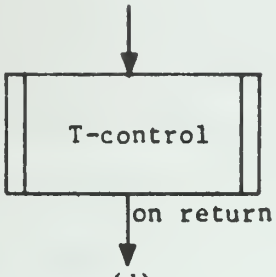
(a)



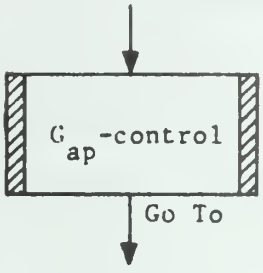
(b)



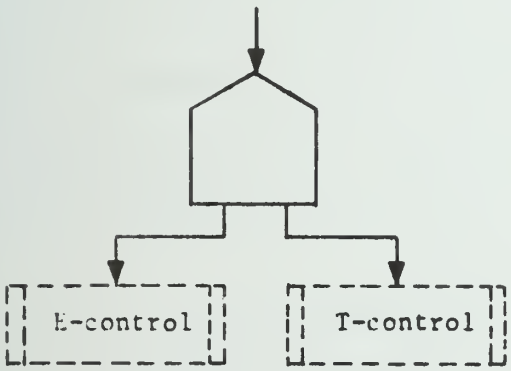
(c)



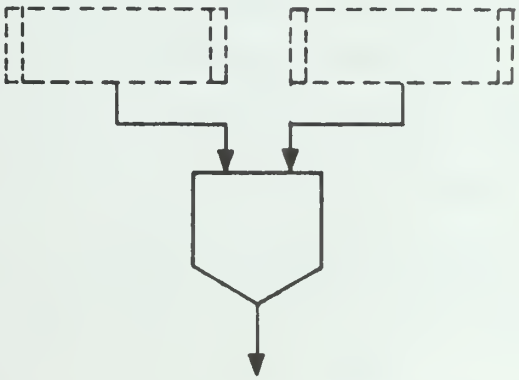
(d)



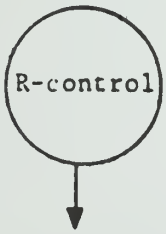
(e)



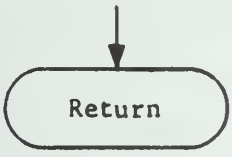
(f)



(g)



(h)



(j)

(k)

Figure 4.21 Control Sequence Chart Symbols.

the subcontrols at the exit points of the symbols be activated concurrently. On the other hand, the 'JOIN' symbol signifies that the replies from all the concurrently active control sequences indicated by the entry points to the box must be true before the control flow can proceed any further.

The entry to a control sequence chart is indicated by a single circle (Figure 4.21h) with the name of the corresponding subcontrol written in the circle. The oval symbol (Figure 4.21j) represents a 'return' to the invoking point of the subcontrol in the control flow. A double circle (Figure 4.21k) represents a branch to the entry point of the subcontrol whose name is written inside the circle.

The control sequence charts which are too big to be fitted on a single page have been drawn on different pages but the entry point on each page is labeled the same. An example is the DM-control.

The microoperations within a control step box are indicated by either control signals of the form

control signal name: = 1 or 0

or transfer statements of the form

x ← y

x ← 1 or 0.

Most of the control signals in DPL are level signals whereas the interface request-response signals are Pulse signals whose leading and trailing edges are used to indicate request, acknowledge and response states. The '1' or '0' on the right hand side indicate the logically 'active' and 'inactive' state respectively. In the case of transfer statements

indicated by the arrow ←, x represents a control status memory element which is set to the state '1' or '0' or to the state of 'y'.

The control signals for the selector networks are of the form XsY where X indicates the input source to the selector network sY. The gate signals for the register is of the form gRegisterName where RegisterName identifies the register which has to be loaded with information.

Square brackets [ ] indicate a subscript value as in ISP notation [42] and thus the address of a register or memory location when these brackets appear after a memory element name. The value of the subscript is written within the square brackets.

The angle brackets < > enclose lists of bit names. For example, if MIR is a register, then MIR<4:0> indicate bits 0 through 4 of register MIR and that the bits in MIR are numbered from right to left in ascending order.

The subscript i, i-1, i+1 on the signal names indicates the index of the PE originating the interface control signal.

4.3.2.3.2 Description of R-control - The function of the R-control in PE_i is to accept for processing and to acknowledge a microinstruction from the adjacent PE_{i-1}, and to invoke the DM-control for the processing of the microinstruction. The control sequence chart for the R-control is shown in Figure 4.22.

The R-control indicates its readiness to PE_{i-1} to accept another microinstruction by the signal $RACK_i$:=1. The R-control in PE_i monitors the request signal TRQ_{i-1} from PE_{i-1}. The active state of TRQ_{i-1} indicates that information on input port MIP_i is valid and R-control (control step RC1) loads the microinstruction into register MIR<10:0>. (It is

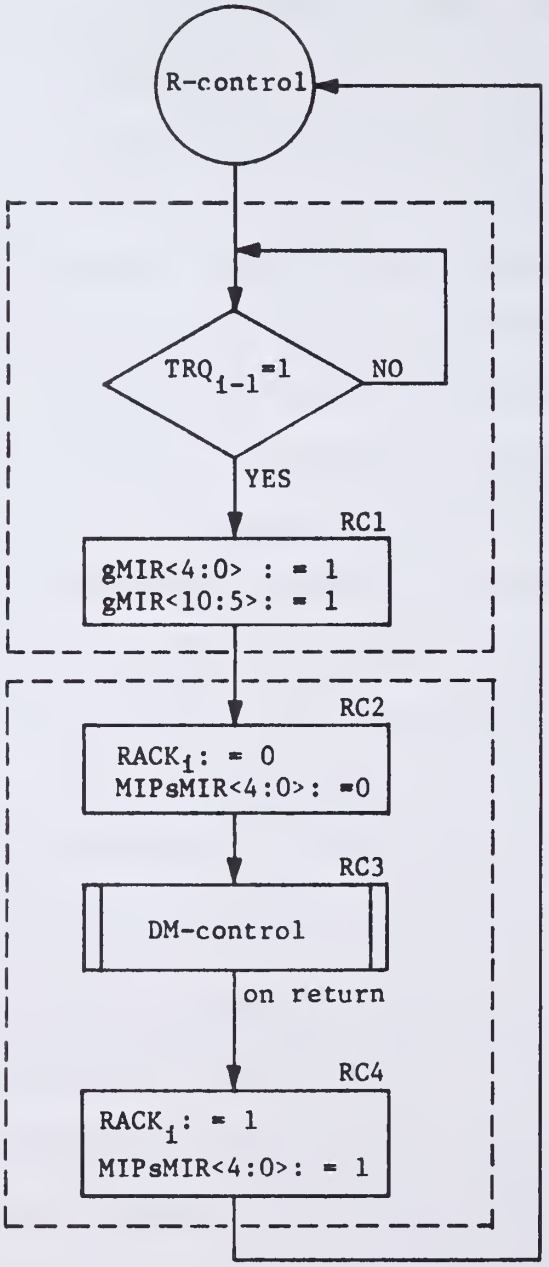


Figure 4.22 Control Sequence Chart for R-control.

assumed that the selector sMIR<4:0> was put earlier in a state to select MIP_i input.) Then the R-control (control step RC2) acknowledges the receipt of the microinstruction by the control signal $RACK_i$:=0. The R-control (control step RC3) then invokes the DM-control for the processing of the microinstruction, and waits for a reply from the DM-control. The reply indicates that the processing is finished and R-control can accept another microinstruction which it (R-control) indicates to PE_{i-1} by the control signal $RACK_i$:=1. At the same time, the selector network sMIR<4:0> is set to select the data from microinstruction input port MIP_i. This is done in control step RC4.

It is assumed, in the control sequence chart of Figure 4.22, that initially, at the power turn on, $RACK_i$:=1 and MIPsMIR<4:0>:=1 are true.

4.3.2.3.3 Description of DM-control - The DM-control can be looked upon as the main control which on being invoked by the R-control monitors the output of the microinstruction decoder. Depending on the nature of the microinstruction, it sets up the necessary data paths and conditions the combinational logic in the data flow logic of the PE. After the data paths are set up, the DM-control invokes one or more of the other controls, F, E, G for processing and T-control, if necessary for onward transmission of the microinstruction to PE_{i+1}. Since the selectors have no memory, and the data paths remain set throughout the processing, the output of the microinstruction decoder can be directly connected to the selector signals of the form XsY: = 1 and involves no extra logic cost. Figures 4.23a, b, and c show the control sequence chart for the DM-control.

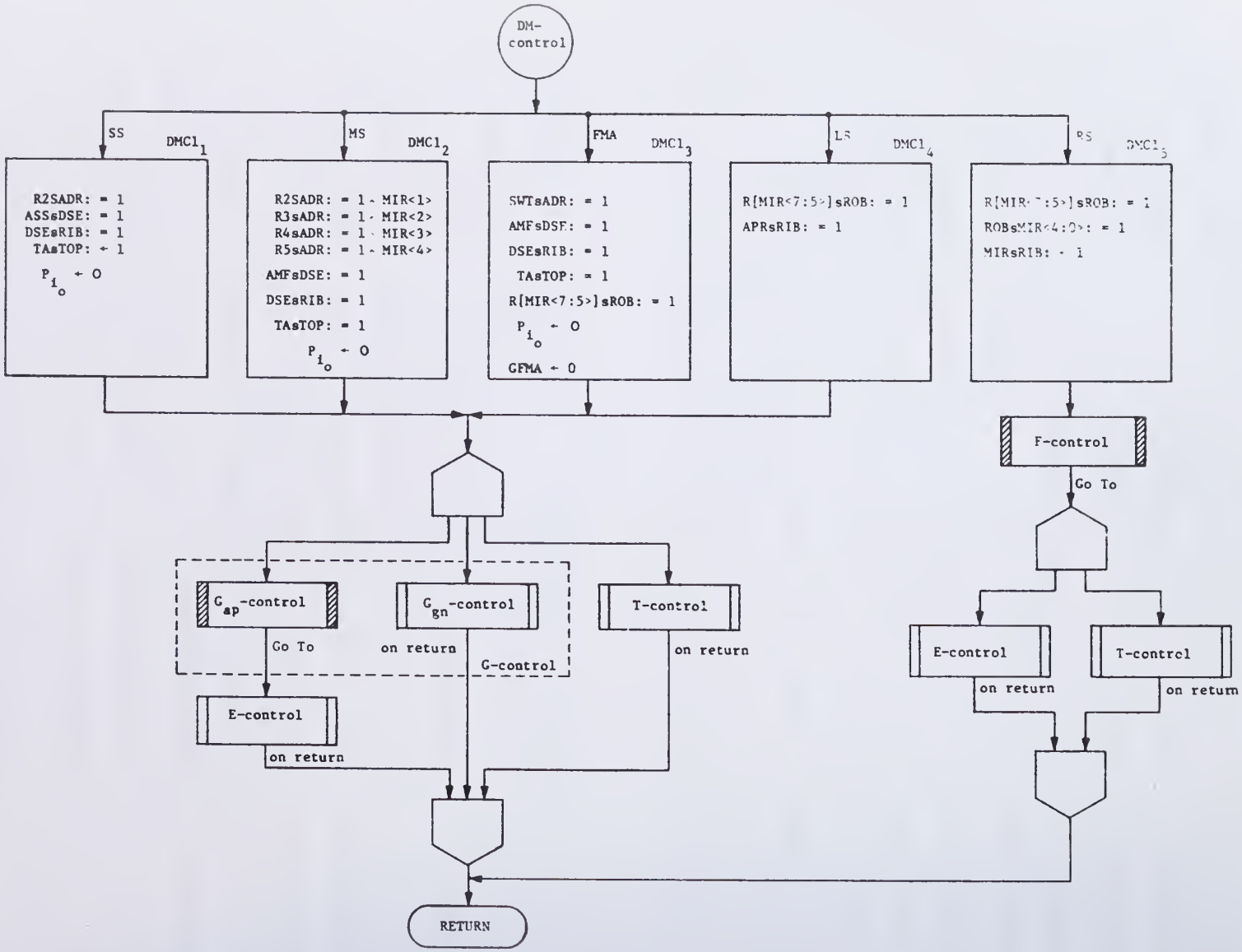


Figure 4.23a Control Sequence Chart for DM-control, Part I.

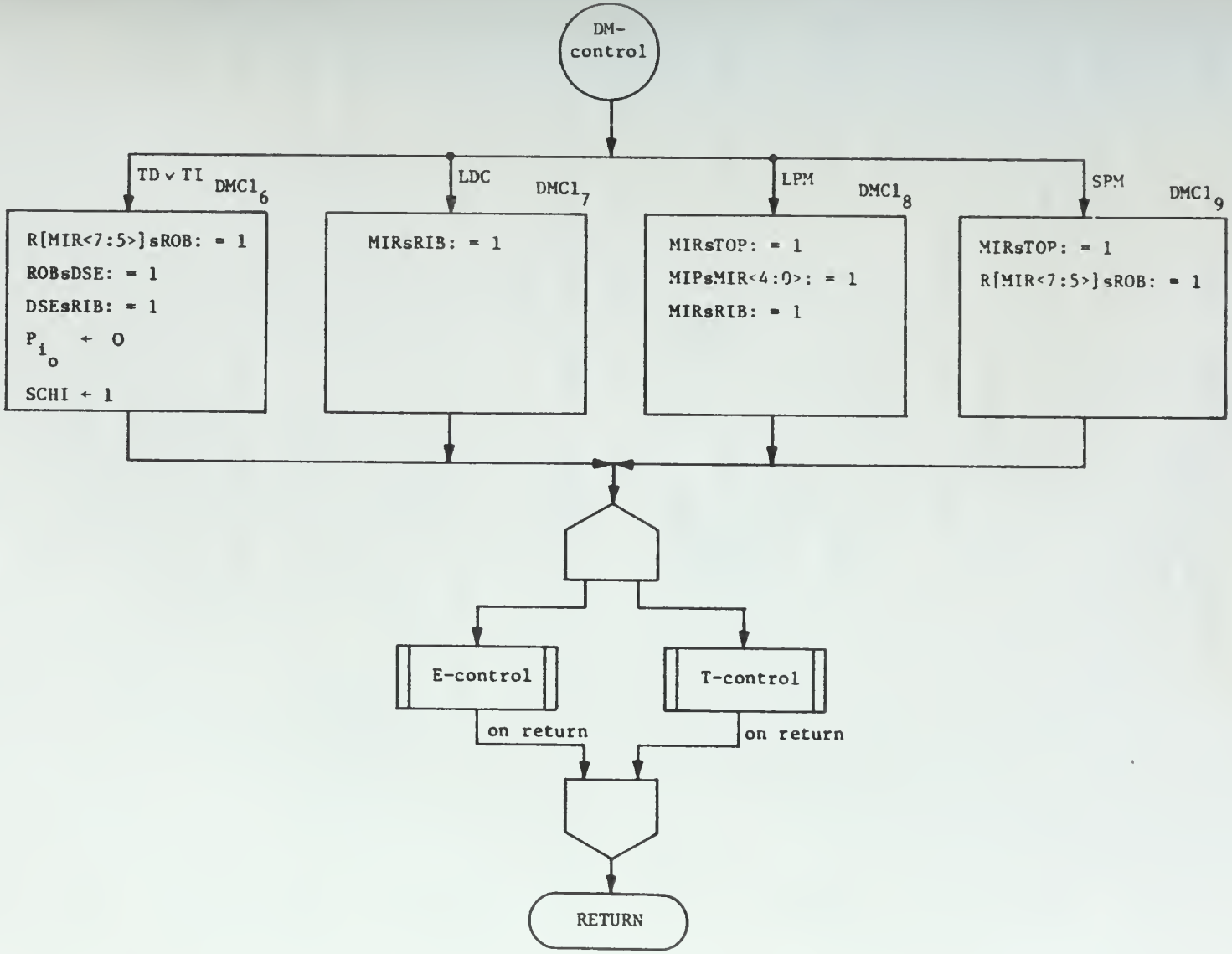


Figure 4.23b Control Sequence Chart for DM-control, Part II.

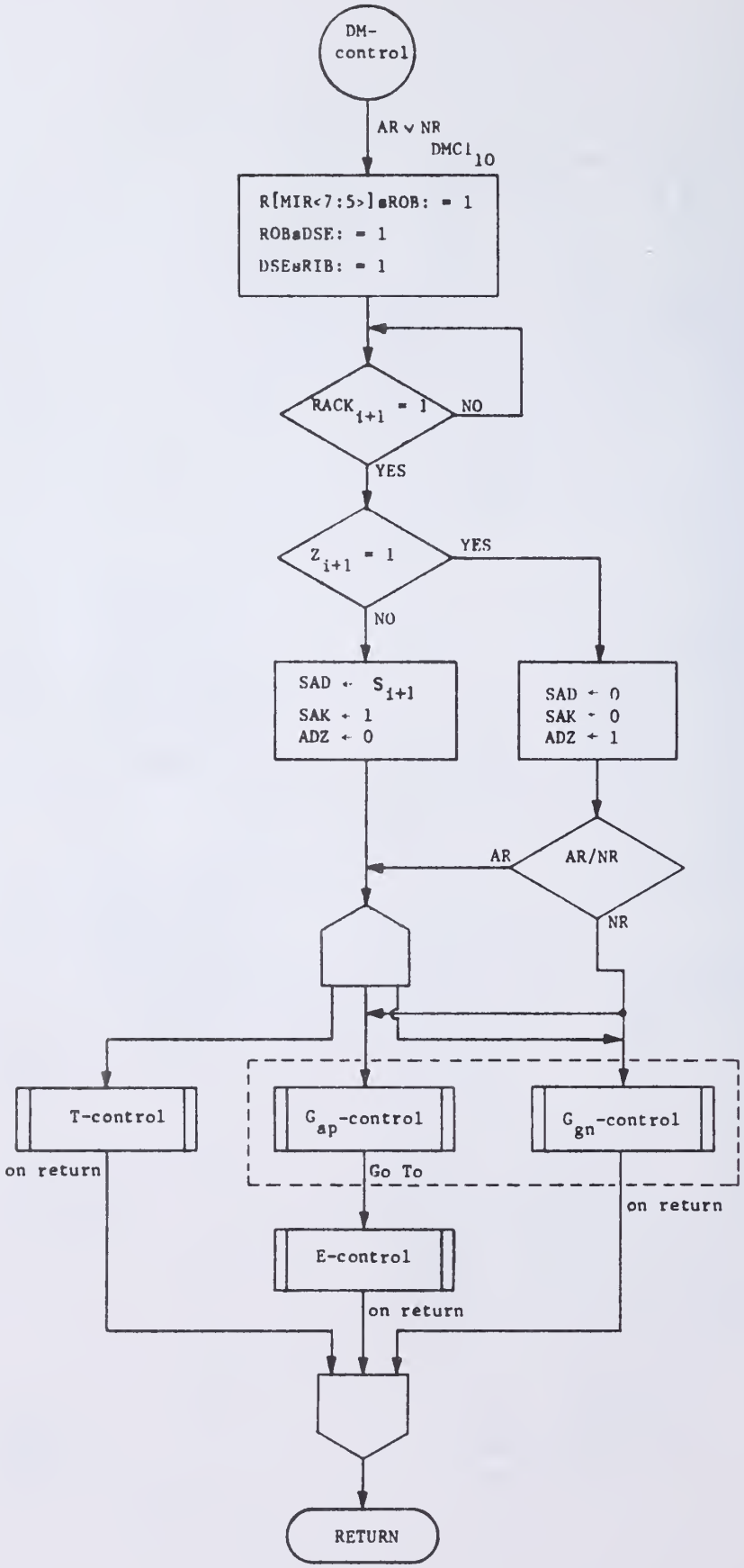


Figure 4.23c Control Sequence Chart for DM-control, Part III.

The data path for the microinstruction SS is through the selector sADR for operand register INR2, through the selector sDSE and encoder DSE for the result (sum of the contents of INR1 and INR2) digit and finally through the selector sRIB. The encoder DSE converts the redundant binary sum digit to sign-magnitude format SM_r. This data path is set up by control step $DMC1_1$. The control signal TAsTOP sets up the data path for the 'Adder Transfer' out of PE_i. The microoperation $P_{i_0} \leftarrow 0$ conditions the DSE encoder logic for proper conversion of the redundant binary result digit into the equivalent sign-magnitude format.

For the microinstruction MS, the control step $DMC1_2$ sets up the selector sADR for the source operands and the data paths for the result digit through the selectors sDSE and sRIB and for the 'Adder Transfer' through the selector sTOP.

If the microinstruction to be processed is FMA, the control step $DMC1_3$ sets up the necessary data paths--for the 'product array' and 'product transfer' through sADR, that of result digit through sDSE and sRIB, of 'Adder Transfer' through sTOP. The multiplicand digit from operand register INR2 is put on the register output port ROP_i via selector sROB. The control memory flip-flop GFMA acts as a synchronizing device between the concurrently active and interacting controls--G_{gn}-control and G_{ap}-control. It is initialized to state '1'. The details of its action are discussed later in Section 4.3.2.3.6.

Control steps $DMC1_4$ through $DMC1_9$ set up the data paths for the microinstruction shown at the entry points of each control step in Figures 4.23a, 4.23b, and 4.23c. For the left shift microinstruction

LS, the data path for the digit in PE_i is from the internal register to the output port ROP_i via the selector sROB and the data path for the incoming digit from PE_{i+1} is from RIP_i to register IBR via the register APR and selector sRIB. In the case of microinstruction RS, the digit to be stored is in microinstruction register MIR<4:0> and its corresponding data path to the input of register file is via the selector sRIB. The data path for the digit to be shifted out to PE_{i+1} is via the selector sROB, the bus ROB and the selector sMIR<4:0> to register MIR and thence to port MOP_i. For the inter-register transfer microinstructions TD and TI, the data path is through the selector sROB, the bus ROB, the selectors sDSE and sRIB. The control memory flip-flop SCHI generates the similar named control signal which transforms the SM_r-encoded output of ROB into redundant binary format for proper transfer. The logical '1' state of control signal SCHI guarantees that the magnitude bits of input digit on ROB will appear unchanged at the output of DSE. This can be seen from Figure 4.14 and the discussion in Section 4.2.2.6. The data path for the microinstruction LDC is from the register MIR through the selector sRIB to the proper register INR[MIR<7:5>] of register file via the buffer register IBR.

For the memory access microinstructions, the communication of data and address takes place via the ports ROP_i, MIR_i, and TOP_i. For the microinstructions SPM and LPM, the data path for the address of the location in PEM and the read/write bit is from MIR to TOP_i via the selector sTOP. However, the data path for the data to be stored in case of SPM is from register INR[MIR<7:5>] through selector sROB to the

output port ROP_i. But the data path from memory to the register INR[MIR<7:5>] for microinstruction LPM, is via port MIP_i, selector sMIR<4:0>, register MIR<4:0>, the selector sRIB and buffer register IBR.

The data path for the microinstructions AR and NR is from the register INR1 through the selectors sROB, sDSE, encoder DSE and the selector sRIB back to INR1 via buffer register IBR. Note that the OP-codes for the microinstructions AR and NR are so chosen that bits MIR<7:5> address the register INR1. This explains the reasons for the OP-code choices, for various microinstructions shown in Figure 4.20.

After the data paths are set up, the DM-control invokes one or more of G, F, E and T-controls for actually processing of the micro-instruction. The microinstructions SS, MS, FMA and LS all require G-information from their right neighboring PEs. So the DM-control invokes the G-control consisting of G_{gn}-control and G_{ap}-control and the T-control in parallel. The T-control transmits the present micro-instruction to PE_{i+1}. The identity of microinstruction in PE_i is essential in PE_{i+1} to generate the G-information for the microinstruc-tion processing. The control flow at the end of G_{ap}-control branches directly to the E-control for the actual calculation and storage of the result digit.

When all the concurrently invoked subcontrols are finished, they report back to the DM-control at the invoking point in control flow. The DM-control now replies back to the R-control which had earlier invoked DM-control. The R-control was in a state of active suspension

(wait state) during the activity of DM-control. The R-control now gets ready to receive another command as explained earlier.

For the microinstruction RS, DM-control, after setting up the data paths, invokes F-control which changes the modifer field of the micro-instruction in MIR and PE_i for transmission to PE_{i+1}. The details of F-control are discussed later. At the end of F-control, E-control and T-control are invoked in parallel but no G-information is required for the processing of microinstruction RS. On return from both the concurrently active E- and T-controls, the DM-control replies back to the waiting R-control.

In the case of microinstructions TD, TI, LDC, LPM and SPM (Figure 4.23b) no G-information is required. Hence DM-control invokes only the E-control and T-control in parallel. The rest of the control flow is as it is for RS.

The invoking of E, G and T-controls by DM-control for microinstructions AR and NR is more complex since it (invocation) depends on the nature of the data resident in the adjacent PE_{i+1}. The digit algorithm of the microinstruction AR discussed in Section 3.6.5, requires knowing the sign of the first non-zero digit to the immediate right of the present digital position. This is done through the use of interface control signals S_i, Z_i and control memory flip-flops SAD, SAK and ADZ which respectively stand for the Sign of Adjacent Digit, Sign of Adjacent Digit Known and Adjacent Digit is Zero. The value of logical '1' for SAK and ADZ indicate assertion or truth whereas '0' indicates falsehood.

The interface control signal S_i (which is the outputs of control memory flip-flop S_i) indicates the sign of the digit in PE_i's accumulator register. Z_i (which is also the output of flip-flop Z_i) indicates whether the magnitude of the accumulator digit is zero. $Z_i = 1$ indicates that the digit is zero and $Z_i = 0$ indicates otherwise. Note that if Z_i is monitored by adjacent PE_{i-1}, validity of Z_i can only be ensured when $RACK_i$=1; i.e., when PE_i is not in the middle of executing any previous microinstruction. The mechanism for determining the sign of the first nonzero digit to the right of the present digital position, i say, is as follows.

If the digit in PE_{i+1} is zero, G_{ap}-control in PE_i goes into a wait loop. In the meantime, the microinstruction AR is passed to PE_{i+1} where again Z_{i+2} is monitored to see if the digit in PE_{i+2} is zero. If it is, it (G_{ap}-control in PE_{i+1}) also goes into a wait loop and the microinstruction passes to PE_{i+2}, PE_{i+3}, ..., PE_{i+j} if $Z_{i+j+1} = 0$ and Z_{i+3}, Z_{i+4}, ..., Z_{i+j} are all in logical state '1'. The G_{ap}-controls in PE_{i+2}, ..., PE_{i+j-1} go into the wait loop. As soon as $Z_{i+j+1} = 0$ is monitored by PE_{i+j}, G_{gn}-control in PE_{i+j} assigns the value of S_{i+j+1} to S_{i+j} and declares the sign valid to the waiting G_{ap}-control in PE_{i+j-1} by assigning logical state '1' to the control signal GRQ_{i+j}. The G_{ap}-control in PE_{i+j-1} informs the G_{gn}-control in PE_{i+j-1} about the validity of sign S_{i+j}, by setting synchronizing control flip-flop SAK, in PE_{i+j-1}, to logical state '1'. The G_{gn}-control in its turn assigns the value of S_{i+j} to S_{i+j-1} and declares the sign S_{i+j-1} to be valid. The sign of the digit thus flows backward till PE_i is reached and in

this way, all the zero digits lying to the immediate right of digital position i are assigned the sign of the first non-zero digit.

We now describe the action of DM-control for microinstructions AR and NR. The DM-control checks the state of control signal Z_{i+1} by monitoring the control signal $RACK_{i+1}$ as explained earlier. If the adjacent digit in PE_{i+1} is not zero ($Z_{i+1} \neq 1$), the control memory element SAD is set to the state of S_{i+1}', the sign of adjacent digit is declared known by SAK $\leftarrow$ 1 and the adjacent digit is declared non-zero by setting ADZ to logical state '0'. However, if $Z_{i+1} = 1$, the control memory flip-flops SAD and SAK are set to logical state '0' and flip-flop ADZ to state '1'.

For the microinstruction AR, the DM-control then invokes T-control and G-control in parallel irrespective of the state of the control signal Z_{i+1}. However, for microinstruction NR, no digit beyond and including the first (counting from left) zero digit of the operand needs to be recoded. So the flow of microinstruction NR stops as soon as $Z_{i+1} = 1$ is encountered. This is done by the DM-control not invoking the T-control in PE_i. However, G_{gn}-control and G_{ap}-control are invoked for uniformity of invoking procedure, although G_{ap}-control is immediately exited for microinstruction NR as can be seen from the control sequence chart for G_{ap}-control in Figure 4.27.

When all the parallely invoked controls have finished, the DM-control replies back to the waiting R-control which gets conditioned to receive another microinstruction for processing in PE_i.

4.3.2.3.4 Description of T-control - The T-control, when invoked by DM-control, passes the microinstruction in register MIR of PE_i to the PE_{i+1}. The control sequence chart for T-control is shown in Figure 4.24. The T-control in PE_i monitors the signal $RACK_{i+1}$ (from the R-control in PE_{i+1}) whose logical state '1' indicates that R-control in PE_{i+1} is ready to accept the microinstruction. The control step TC1 sets the control memory flip-flop whose output gates the contents of MIR onto bus MOP_i. Then in control step TC2, the request signal TRQ_i is activated which in turn is monitored by the R-control in PE_{i+1}. As soon as R-control in PE_{i+1} accepts the microinstruction from MOP_i ($\equiv MIP_{i+1}$), it (R-control) acknowledges by assigning the '0' logical state to acknowledge signal $RACK_{i+1}$. The '0' state of $RACK_{i+1}$, being monitored by T-control, signifies that the microinstruction has been accepted and then the control step TC3 withdraws the request for transmission by assigning '0' logical state to request signal TRQ_i and also removes the information from the bus MOP_i (MIP_{i+1}). The latter is necessary for microinstruction LPM where the port MIP_{i+1} is used for inputing data read from the PEM_i.

At the end of the control sequence, the control flow returns to the point where the t-control was invoked.

4.3.2.3.5 Description of F-control - The function of the F-control is to modify the microinstruction modifier field ${}_jF_i$ before transmission of the microinstruction to the next PE, i.e., PE_{i+1}. This is made use of in the microinstruction RS where the modifier field carries the digit

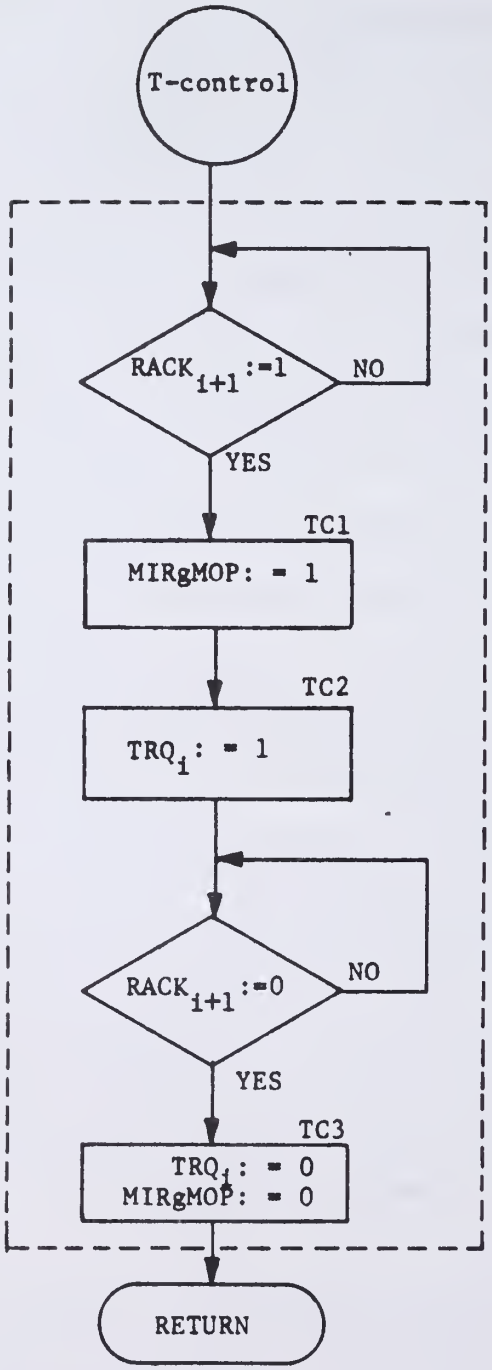


Figure 4.24 Control Sequence Chart for T-control.

to be shifted into the adjacent PE. Figure 4.25 shows the control sequence chart for F-control. Control step FC1 loads the buffer register IBR from the output of the selector sRIB. The selector sRIB was initially conditioned by control step $DMC1_5$ in DM-control to select the digit from MIR<4:0>. At this time, the MIR<4:0> carries the digit from the adjacent PE_{i-1} and it arrived as part of the microinstruction from PE_{i-1}. Control step FC2 loads the MIR<4:0> from the output of the selector sMIR<4:0> which was conditioned to accept the digit from INR[MIR<7:5>] in PE_i to be shifted into next PE_{i+1} by control step $DMC1_5$. At the end of control step FC2, the control flow branches to initiate E-control and T-control in parallel. The T-control would transmit the microinstruction in MIR with the new modifier field and the E-control would load the register INR[MIR<7:5>] from the buffer register IBR.

4.3.2.3.6 Description of G-control - The G-control consists of two independent subcontrols: G_{gn}-control which generates the G-information (mainly 'Adder Transfer' t^A_{i-1}) for the adjacent PE_{i-1}; and G_{ap}-control which accepts the G-information from the adjacent PE_{i+1}. The G-control is invoked by DM-control only when the processing of the microinstruction requires information from the adjacent PEs. When the G-information depends logically on more than one adjacent PE, the G_{gn}-control and G_{ap}-control interact with each other through synchronizing control memory flip-flops GFMA (in the case of microinstruction FMA) and SAK (for the microinstructions AR and NR).

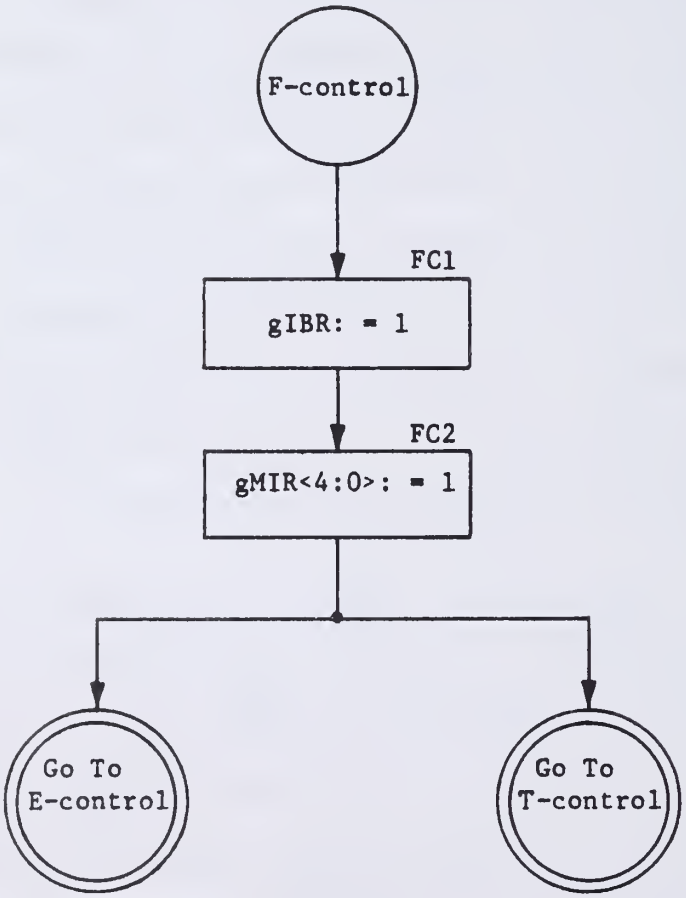


Figure 4.25 Control Sequence Chart for F-control.

4.3.2.3.6.1 Description of G_{gn}-control - The function of G_{gn}-control is to generate G-information needed in the adjacent PE_{i-1}. Figure 4.26 shows the control sequence chart for this control. The G-information for the microinstructions SS and MS consists of the 'Adder Transfer' t^{A}_{i-1} which is routed to the output port TOP_i by the control steps $DMC1_1$ and $DMC1_2$. The G-information for the microinstruction LS is the digit in register INR[MIR<7:5>] which is routed to port ROP_i by data path set up in control step $DMC1_4$. After the 'G'-information stabilizes on ports TOP_i and ROP_i, control step Ggn1 informs the G_{ap}-control in PE_{i-1} about the validity of G-information. The G_{gn}-control then monitors the acknowledgment signal $GACK_{i-1}$ from G_{ap}-control of PE_{i-1}. When $GACK_{i-1}$ is in logical state '1', the control step Ggn2 declares the G-information not valid.

For the microinstruction FMA, the G-information ${}_{FMA}G_i$ to be generated by PE_i for PE_{i-1} consists of 'Adder Transfer' t^{A}_{i-1} and the multiplicand digit ϕ_i (assuming 'local generation' of CPT t^{P}_{i-1}). The 'Adder Transfer' t^{A}_{i-1} is dependent on the multiplicand digit ϕ_i and accumulator digit a_i in registers INR2 and INR1 respectively in PE_i, multiplicand digit ϕ_{i+1} and accumulator digit a_{i+1} in registers INR2 and INR1 respectively in PE_{i+1} and the multiplier digit m_j. t^{A}_{i-1} consists of two parts t^{A0}_{i-1} and t^{A1}_{i-1} and is generated in a time sequential manner. The process of generation of ${}_{FMA}G_i$ can be represented in the notation of Section 2.4 as follows:

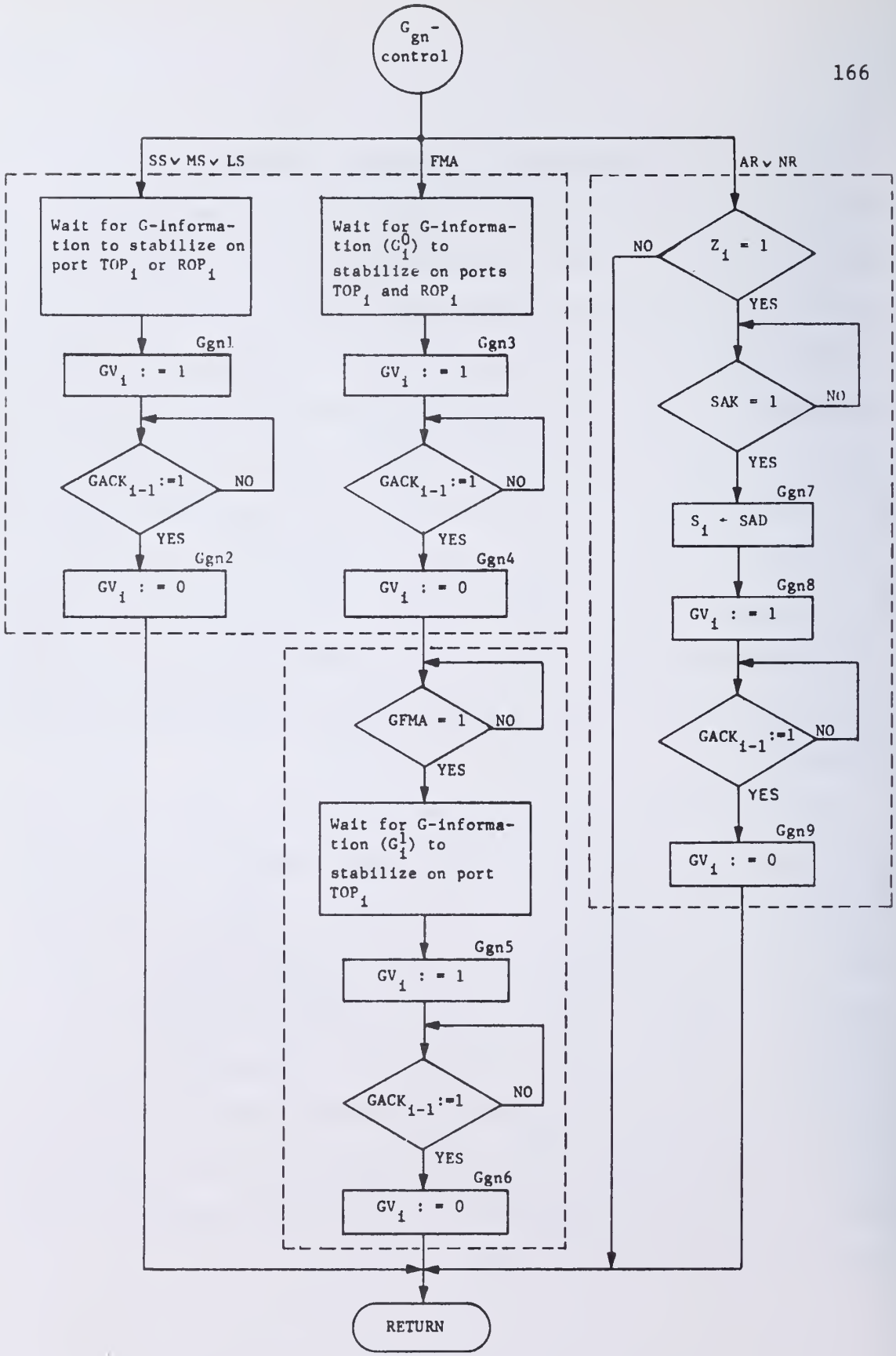


Figure 4.26 Control Sequence Chart for G_{gn}-control.

$$_{FMA}G_i = \{t^{A}_{i-1}\ ,\ \phi_i\}$$

$$t^{A}_{i-1} = \Gamma(m_j,\ \phi_i,\ a_i,\ \phi_{i+1},\ a_{i+1})$$

$$= \{t^{A0}_{i-1}\ ,\ t^{A1}_{i-1}\}$$

where $$t^{A0}_{i-1} = \Gamma^0(m_j,\ \phi_i,\ a_i)$$

$$t^{A1}_{i-1} = \Gamma^1(m_j,\ \phi_i,\ a_i,\ \phi_{i+1},\ t^{A0}_i)$$

and $$_{FMA}G_i = \{t^{A0}_{i-1},\ t^{A1}_{i-1},\ \phi_i\}$$

$$= \{G^0_i\ ,\ G^1_i\}$$

where $$G^0_i = \{t^{A0}_{i-1},\ \phi_i\}$$

and $$G^1_i = \{t^{A1}_{i-1}\}\ .$$

In the above relations, the subscript FMA has been dropped from all variables for ease of reading. The above described structure for G^0_i and G^1_i can be deduced from an examination of Figures 3.13 and 4.11d together.

When the G-information G^0_i is valid on ports TOP_i and ROP_i which carry the t^{A0}_{i-1} and ϕ_i components of G^0_i respectively, control step Ggn3 informs PE_{i-1} about the validity of G^0_i information. After PE_{i-1} has accepted (indicated by $GACK_{i-1}$:=1) the G^0_i, control step Ggn4 sets validity signal GV_i to logical state '0'. To generate G^1_i, it is necessary that G^0_{i+1} (= $\{t^{A0}_i,\ \phi_i\}$) be available in PE_i. When G^0_{i+1} from PE_{i+1} is valid on input ports TIP_i and RIP_i, the G_{ap}-control in PE_i accepts

and stores G^0_{i+1} and informs the G_{gn}-control about its availability by setting to logical state '1' the control memory flip-flop GFMA. As soon as the synchronizing flip-flop GFMA is in logical state '1' and G^1_i is valid on port TOP_i ($G^1_i = t^{A1}_{i-1}$ is automatically generated by the logic in MIAD of PE_i), the control steps Ggn5 and Ggn6 declare the G^1_i information valid and invalid respectively in the same way as do control steps Ggn3 and Ggn4.

For the microinstructions AR and NR, the G-information consists of only the sign of the digit in the accumulator of adjacent PE_{i+1} and also whether the digit is zero or not. If the digit in the accumulator in PE_i is non-zero ($Z_i \neq 1$), then no G-information needs to be generated because it is already known to the PE_{i-1} via its DM-control. However, if the present digit is a zero ($Z_i = 1$), the meaningful sign for this zero digit is the sign of the first non-zero digit to its right. If the digit to the immediate right in PE_{i+1} is non-zero, then the sign of the adjacent digit is known and is stored in flip-flop SAD by DM-control in PE_i earlier. If, however, the digit in PE_{i+1} is zero ($Z_{i+1} = 1$), the sign of the adjacent digit is unknown ($SAK \neq 1$). The G_{gn}-control goes into a wait loop continuously monitoring SAK till the G_{ap}-control in PE_i determines and stores the sign in SAD of the digit in PE_{i+2}. As soon as the sign of the adjacent digit is known, control step Ggn7 assigns the same value to the flip-flop S_i whose value represents the sign of accumulator digit in PE_i. Control step Ggn8 informs the G_{ap}-control in PE_{i-1} about the validity of S_i. After G_{ap}-control in PE_{i-1} acknowledges the receipt of valid sign S_i ($GACK_{i-1} = 1$), control step Ggn9 withdraws the validity signal. G_{gn}-control now returns to the invoking point in DM-control.

4.3.2.3.6.2 Description of G_{ap}-control - The function of G_{ap}-control is to accept the G-information generated by G_{gn}-control in PE_{i+1}. Figure 4.27 shows the control sequence chart for G_{ap}-control. This G-information is available on port TIP_i (for microinstructions SS, MS and FMA), on port RIP_i (for microinstructions FMA and LS) and on interface control line S_{i+1} (in case of microinstructions AR and NR).

In the case of microinstructions SS and MS, the G_{ap}-control monitors the validity interface signal GV_{i+1}. As soon as the G-information is valid, control step Gap1 stores the G-information on bus TIP_i into G-information register GIR. Control step Gap2 acknowledges the receipt of G-information, and control step Gap3 withdraws the acknowledgment signal $GACK_i$ once the validity signal is withdrawn ($GV_{i+1} = 0$) by G_{gn}-control in PE_{i+1}. For microinstruction LS, the same sequence is followed except that G-information is available on RIP_i and is stored in register APR by control step Gap4.

As explained earlier, G-information G_{i+1} for microinstruction FMA consists of two components: $G^0_{i+1}(=\{t_i^{A0}, \phi_{i+1}\})$ and $G^1_{i+1}(=\{t_i^{A1}\})$. When the G^0_{i+1} information is valid, the control step Gap7 stores t_i^{A0} component in register GIR and ϕ_{i+1} component in register APR. Then control step Gap8 sets the synchronizing flip-flop GFMA to logical state '1' to inform the G_{gn}-control about the availability of G^0_{i+1} information so that G_{gn}-control may generate G^1_i for PE_{i-1}. Control steps Gap9 and Gap10 play the same role of acknowledgment assertion and its withdrawal as control steps Gap2 and Gap3. After control step Gap10, G_{ap}-control again starts monitoring the validity control signal for G^1_{i+1}. As soon

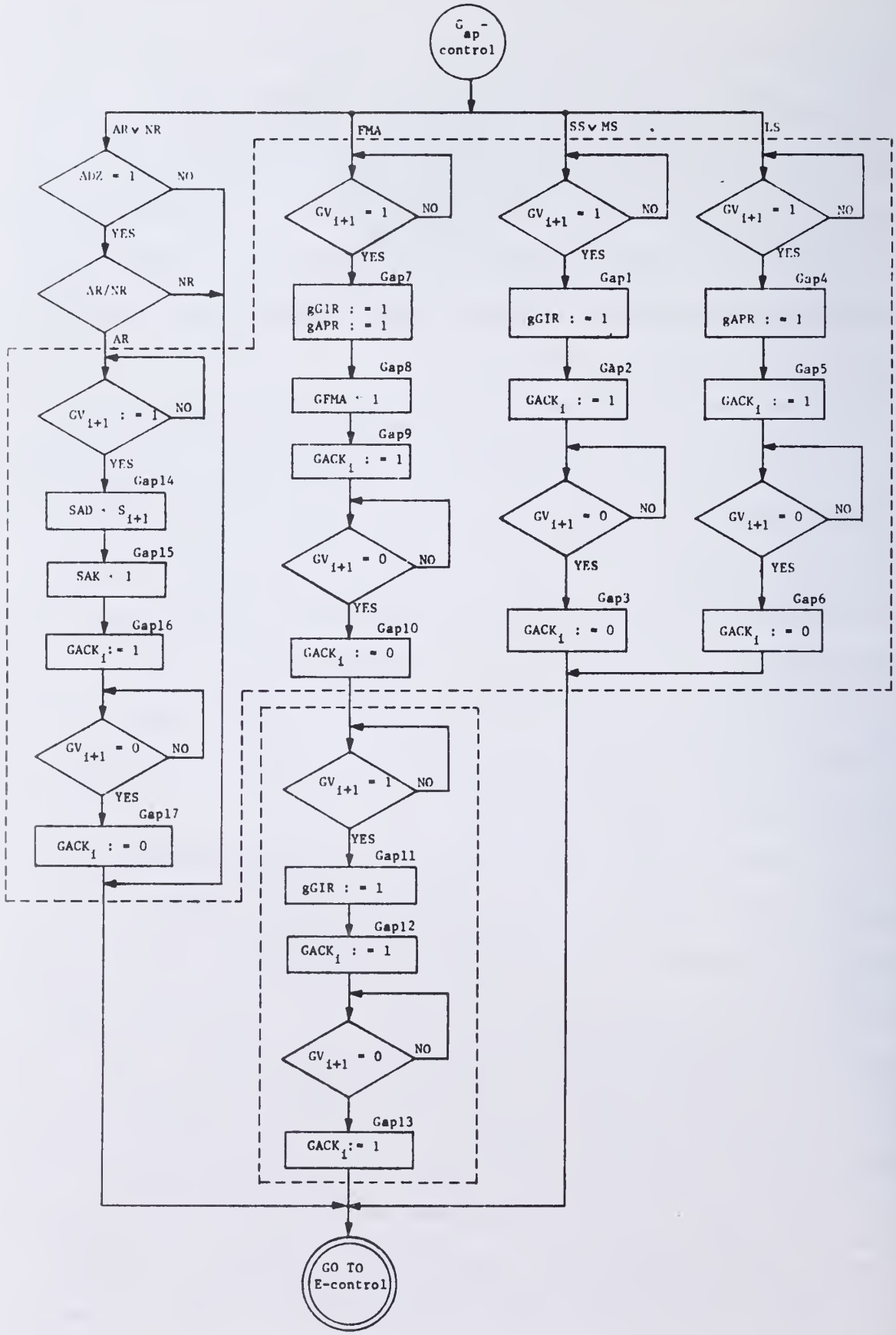


Figure 4.27 Control Sequence Chart for G_{ap}-control.

as G^{1}_{i+1} is valid (GV_{i+1} = 1) on TIP_i, control step Gap11 stores it in G-information buffer register GIR and then control steps Gap12 and Gap13 respectively acknowledge the receipt of G^{0}_{i+1} information and withdraws the acknowledge signal on response from G_{gn}-control.

For the microinstructions AR and NR, if the adjacent digit is non-zero or if it is zero and the microinstruction is NR, no G-information needs to be accepted from adjacent G_{gn}-control and the G_{ap}-control is immediately exited.

In the case of zero adjacent digit and microinstruction AR, G_{ap}-control monitors the G-information validity signal. Here the G-information consists of S_{i+1}. As soon as G_{gn}-control has determined the valid sign for S_{i+1} (which is the sign of first non-zero digit to its right), G_{gn}-control sets the validity signal GV_{i+1} to logical state '1'. As soon as G_{ap}-control in PE_{i+1} finds GV_{i+1} in state '1', the control step Gap14 stores the sign S_{i+1} in flip-flop SAD and Gap15 sets the synchronizing flip-flop SAK to logical state '1'. (SAK is being monitored by G_{gn}-control in PE_i in order to attach this sign (stored in SAD) to S_i.) Control steps Gap16 and Gap17 play the same role as Gap2 and Gap3.

At the end of execution of G_{ap}-control, the control sequence for the processing of the microinstruction branches directly to E-control where the result digit is calculated and stored in the appropriate register of the register file.

4.3.2.3.7 Description of E-control - Figure 4.28 shows the control sequence chart for E-control. For the microinstructions SS, MS, FMA, LDC and LS, the E-control loads the result digit, which is available at the

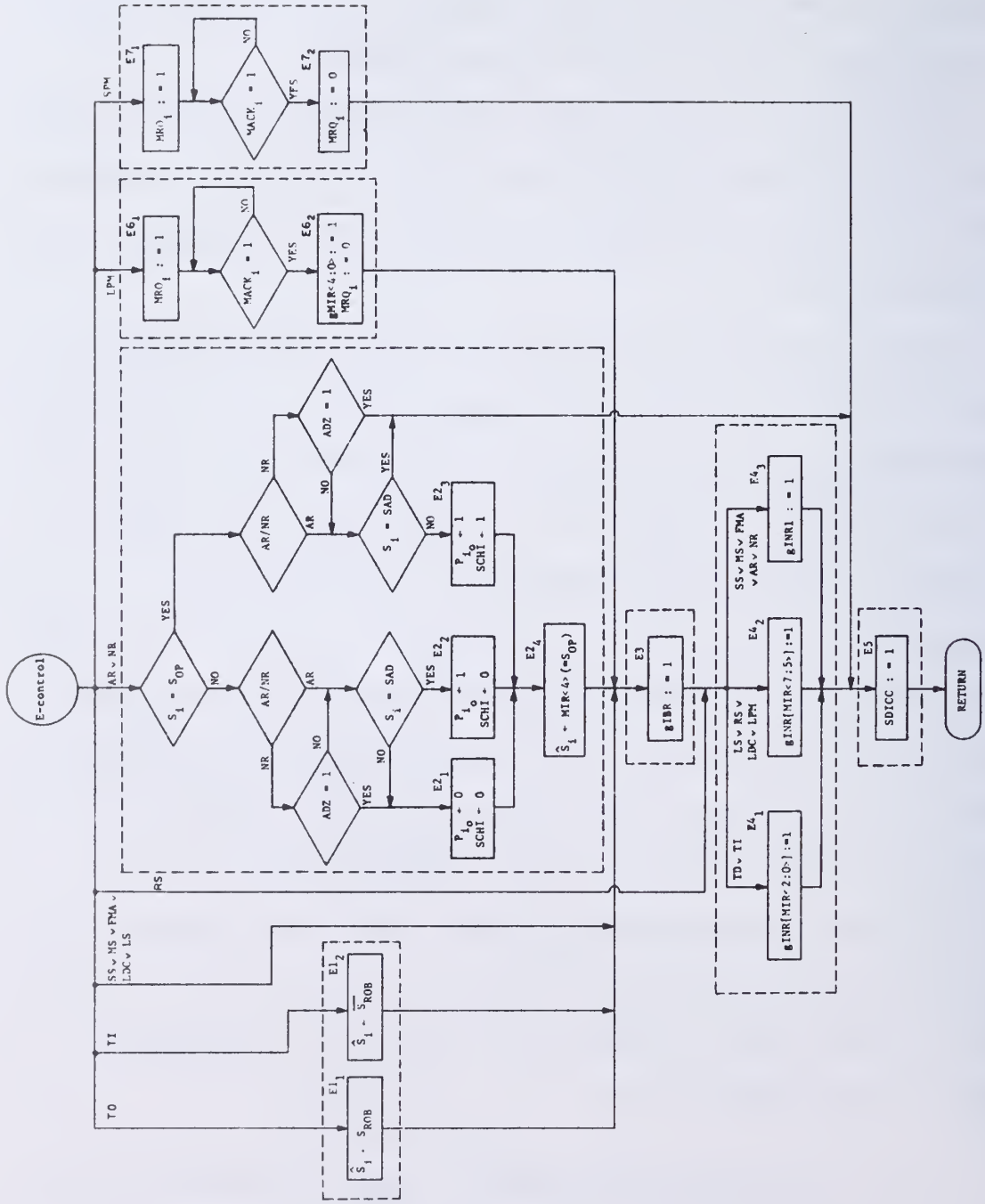


Figure 4.28 Control Sequence Chart for E-control.

output of selector sRIB, into buffer register IBR. This is done in control step E3. Then control steps $E4_3$ and $E4_2$ transfer the contents of IBR into accumulator register INR1 for microinstructions SS, MS, FMA and into the destination register INR[MIR<7:5>] for LDC and LS. Finally, the control step E5 sets the status indicators S_i and Z_i.

For the microinstruction RS, the control step E3 is bypassed and control step $E4_2$ loads the register to be shifted. E5 sets the digit status indicators.

For the inter-register transfer microinstructions, the state of the sign bit S_{ROB} of the digit on the bus ROB is transferred to the sign bit output $\hat{S}_i$ of digit sum encoder DSE for TD. The complement of the state of S_{ROB} is transferred in the case of microinstruction TI. This is done in control steps $E1_1$ and $E1_2$ respectively. The control sequence then goes through the control steps E3, $E4_1$ and E5. Control step $E4_1$ loads the destination register in the register file.

For the microinstruction LPM, control step $E6_1$ requests access to the local memory PEM_i of the PE_i. Note that the address of the location in PEM_i and the Read/Write bit (in the state 'Read') is already available on the output port TOP_i. The PEM_i reads out the data on the micro- in PEM and the Read/Write bit (in state Read) is already available on the output port TOP_i. The PEM_i reads out the data on the micro-instruction input bus MIP_i and informs the PE_i by the logical state '1' of acknowledge signal $MACK_i$. The control step $E6_2$ loads the register MIR<4:0> from the output of selector sMIR<4:0> which had been earlier conditioned, in DM-control, to accept this output and also withdraws the

request for memory access. The control steps E3 and $E4_2$ load the buffer register IBR and file register INR[MIR<7:5>]. Finally the control sequence goes through control step E5 for setting the status indicators.

For the store microinstruction SPM, the address of the PEM_i location is already available on output bus TOP_i and the digit to be stored is on bus ROP_i, when the control flow enters E-control. Control step $E7_1$ requests access to the memory. Then the PEM_i responds by the logical state '1' of acknowledge signal $MACK_i$, after accepting the data and address from the buses TOP_i and ROP_i. Now control step $E7_2$ withdraws the request for memory access. The control sequence finally goes through status setting control step E5.

For the microinstructions NR and AR, the E-control implements the digit algorithms discussed earlier in Sections 3.6.4 and 3.6.5. Control steps $E2_1$ and $E2_2$ respectively achieve the radix complement and the diminished radix complement of the magnitude bits of the accumulator digit. Control step $E2_3$ diminishes the magnitude of the accumulator digit by unity. The particular setting of control signals to states shown in control steps $E2_1$, $E2_2$, and $E2_3$ was explained earlier in Section 4.2.2.6. Control step $E2_4$ assigns the state of MIR<4>, which is the sign, S_{OP}, of the whole operand to be assimilated or normalized, to the sign bit output $\hat{S}_i$ of digit sum encoder DSE. Control steps E_3, $E4_3$ load the result digit in buffer IBR and accumulator register INR1. Finally the control step E5 sets the status indicators regarding the sign and magnitude of the accumulator digit in PE_i.

When the control sequence corresponding to E-control is finished, the control flow returns to the invoking point where G_{ap}-control was invoked in DM-control. This is because the control flow had branched into E-control at the end of the G_{ap}-control sequence.

4.4 Logic Complexity of Processing Element

From the viewpoint of LSI implementation of a PE, two things are of major importance: the number of circuit elements and the number of external pins required for the chip. The total number of circuit elements and pins determine the silicon real estate, density of the circuit elements and the heat dissipation, etc. The number of circuit elements depend on the technology used for the implementation of the logic on the chip. In this thesis, we shall use the number of gates as an indirect measure of logic complexity because the number of circuit elements are directly related to the number of gates. Further, a multi-input NAND gate is considered equivalent to a 2-input NAND gate because in TTL logic, a multi-input NAND is realized by the use of a multi-emitter transistor. These assumptions have been made for the sake of simplicity.

The overall gate complexity and pin complexity of a PE must take into account the gates and pins required by a PE's major components: DPL, PE control logic and Register File.

4.4.1 Logic complexity of DPL

4.4.1.1 Gate complexity of digit processing logic DPL - The total number of gates required for the DPL is equal to the sum of the gates necessary for its various components: Adder MIAD, Digit Product

Generator DPG, Digit Sum Encoder DSE, various selector networks sADR, sDSE, sROB, sRIB, and sTOP and the storage buffer registers in the DPL. The gates required for MIAD, DPG, DSE and selectors sADR and sDSE are dependent on the choice of the logic vector encoding for the redundant binary digit. From the earlier discussion in Sections 4.2.2.2 through 4.2.2.6, it is clear that logic vector encoding LVE_3 is the simplest encoding and requires the least number of gates for the implementation of DPG, sADR and sDSE. In the following, we shall calculate the gate complexity of DPL, assuming only the sign-magnitude (SM_b) logic vector encoding LVE_3.

Let

G_{DPL} = Total number of gates required for DPL, excluding storage registers,

G_{DPG} = Gates required for the logic implementation of Digit Product Generator, DPG, using 'local generation' of t^P_i,

G_{MIAD} = Gates required for the radix-2^k adder, MIAD,

G_{DSE} = Gates required for Digit Sum Encoder, DSE,

and let G_{sDSE}, G_{sRIB}, G_{sROB}, G_{sADR} and G_{sTOP}, respectively denote the gates required for the selectors sDSE, sRIB, sROB, sADR and sTOP. From the design details described in Section 4.2, it is clear that

$G_{MIAD} = 26K^2$ NANDs, assuming no encoder MATE for t^A_{i-1}

$G_{DPG} = K^2$ ANDs + 2 Exclusive-OR gates for sign generation

$= K^2 + 8$ gates

considering a AND and NAND gate equivalent, and 1 Exclusive-OR gate equivalent to 4 NAND gates. From Equations (4.17) through (4.22), we have

$$G_{DSE} = 16K + c_1$$

$$G_{sADR} = 3K^2 + 3K + 1$$

$$G_{sDSE} = 7K$$

$$G_{sRIB} = 4(K + 1)$$

$$G_{sROB} = (K + 1)\ (K + 2)$$

$$G_{sTOP} = 3(K + 2)$$

Therefore, the total number of gates required for the combinational processing logic DPL is given by

$$G_{DPL} = G_{MIAD} + G_{DPG} + G_{DSE} + G_{sADR} + G_{sDSE} + G_{sRIB} + G_{sROB} + G_{sTOP}$$

$$= 26K^2 + (K^2 + 8) + (16K + c_1) + (3K^2 + 3K + 1) + 7K + 4(K + 1) + (K + 1)\ (K + 2) + 3b.$$

Ignoring the constant c_1 and assuming the width b of port TOP_i to be equal to (K + 2), we have

$$G_{DPL} = 31\ K^2 + 36K + 21 \qquad (4.24)$$

In the expression above for G_{DPL}, the sum of the gates contributed by the three major components DPG, MIAD and DSE alone is $(27K^2+16K+8+c_1)$ and forms the bluk of the gates required for the implementation of DPL. The other components like selector networks contribute progressively smaller and smaller percentage of gates to the gate complexity of DPL, as the value of K increases. Table 4.2 lists the values of the gates

Table 4.2 Gate Complexity of DPL vs Radix for $\frac{1}{2} \leq \delta \leq 1$ and LVE_3 Encoding for a Redundant Binary Digit

radix=r	$K=\log_2^r$	G_{DPG}	G_{MIAD}	G_{sADR}	G_{DSE}	G_{sDSE}	G_{sRIB}	G_{sROB}	G_{sTOP}	G_{DPL}
4	2	12	104	19	32	14	12	12	12	217
8	3	17	234	37	48	21	16	20	15	408
16	4	24	416	61	64	28	20	30	18	661
32	5	33	650	91	80	35	24	42	21	976
64	6	44	936	127	96	42	28	56	24	1353
128	7	57	1274	169	112	49	32	72	27	1792
256	8	72	1664	217	128	56	36	90	30	2303

required for various components of the DPL as a function of radix-r and the last column in the table shows the gate complexity of the combinational processing logic.

4.4.1.2 Pin complexity of DPL - Pin complexity is independent of the logic vector encoding chosen for a redundant binary digit. The pins required for digit processing logic DPL is the sum of the pins necessary for input ports TIP_i, RIP_i and output ports TOP_i and RIP_i. Pins, P_{RIP_i} and P_{ROP_i} required for ports RIP_i and ROP_i, respectively are dependent on the method of generation of 'product transfer' t^P_i and t^P_{i-1}, respectively. Similarly, P_{TIP_i}, the pins required for port TIP_i is dependent on the method of encoding used for the 'Adder Transfer' t^A_i. The port TOP_i is shared both by the Read/Write and address lines for PEM_i and the 'Adder Transfer' t^A_{i-1}. P_{TOP_i}, the pins required for TOP_i is the larger of the number of pins required for t^A_i and PEM_i address lines. The total number of pins, P_T necessary for the logic implementation of DPL is equal to the sum of the pins required for input and output ports.

$$P_T = P_{TOP_i} + P_{TIP_i} + P_{RIP_i} + P_{ROP_i}$$

Let

$P^A_{t^P_{i-1}}$ = Pins required for generating t^P_{i-1} using 'Adjacent Generation' method.

$P^L_{t^P_{i-1}}$ = Pins required for generating t^P_{i-1} using 'Local Generation' method.

$P^R_{t^P_{i-1}}$ = Pins required for t^P_{i-1} using ROM for DPG.

$P^{NE}_{t^A_{i-1}}$ = Pins required for t^A_{i-1} without encoder MATE.

$P^{E}_{t^A_{i-1}}$ = Pins required for t^A_{i-1}, using encoder MATE.

$P^{R}_{t^A_{i-1}}$ = Pins required for t^A_{i-1}, using ROM for DPG.

If the Read Only Memory (ROM) is used for the implementation of DPG, and P^R_T is the total number of pins required for DPL, using ROM for DPG, then

$$P^R_T = \text{Max}\,(k+2,\ P^R_{t^A_{i-1}}) + P^R_{t^A_i} + P^R_{t^P_i} + P^R_{t^P_{i-1}}$$

$$= \text{Max}\,(k+2,\ 4) + 4 + (k+1) + (k+1)$$

$$= (k+2) + 4 + (k+1) + (k+1)$$

$$= 3k + 8 \qquad (4.25)$$

Another interesting configuration for the DPL implementation is when DPG uses the 'Local Generation' method for t^P_i and the encoder MATE is used to reduce the pins required for t^A_{i-1}. If P^{EL}_T denotes the total pins required for such a configuration, then

$$P^{EL}_T = \text{Max}\,(k+2,\ P^E_{t^A_{i-1}}) + P^E_{t^A_i} + P^L_{t^P_i} + P^L_{t^P_{i-1}}$$

$$= \text{Max}\,(k+2,\ 2\lceil\log_2(k+1)\rceil) + 2\lceil\log_2(k+1)\rceil + (k+1) + (k+1)$$

$$= (k+2) + 2\lceil\log_2(k+1)\rceil + 2(k+1)$$

$$= 3k + 4 + 2\lceil\log_2(k+1)\rceil \qquad (4.26)$$

Still another implementation configuration of interest for comparison purposes is the one which uses no encoder for the 'Adder Transfer' and the 'Adjacent Generation' method is used for collective product transfer t_i^P. Let P_T^{NEA} be the total number of pins required for such a configuration. Now

$$P_T^{NEA} = \text{Max}(k+2, P^{NE}_{t^A_{i-1}}) + P^{NE}_{t^A_i} + P^{A}_{t^P_i} + P^{A}_{t^P_{i-1}}$$

$$= 2k + 2k + (\frac{k(k-1)}{2} + 1) + (\frac{k(k-1)}{2} + 1)$$

$$= 4k + k(k-1) + 2$$

$$= k^2 + 3k + 2 \qquad (4.27)$$

Finally, we have a configuration which uses no encoder for t^A_{i-1} and the 'Local Generation' method for t_i^P. The total pins P_T^{NEL} for this configuration is given by the following:

$$P_T^{NEL} = \text{Max}(k+2, P^{NE}_{t^A_{i-1}}) + P^{NE}_{t^A_i} + P^{L}_{t^P_i} + P^{L}_{t^P_{i-1}}$$

$$= 2k + 2k + (k+1) + (k+1)$$

$$= 6k + 2 \qquad (4.28)$$

Values of Equations (4.25),(4.26),(4.27) and (4.28) are tabulated in Table 4.3 for various values of the parameter k. It shows that the configuration using ROM for DPG requires the least number of pins. However, the bit capacity ($=k2^{2k+1}$) of ROM required for values of $k \geq 6$ becomes too large and hence not suitable. However, 'Local Generation' of t_i^P and

Table 4.3 Pin Complexity of DPL Vs Radix for $\frac{1}{2} \leq \delta \leq 1$

radix t	$r = 2^k$ k	P_T^R	P_T^{EL}	P_T^{NEA}	P_T^{NEL}
4	2	14	14	12	14
8	3	17	17	20	20
16	4	20	22	30	26
32	5	23	25	42	32
64	6	26	28	56	38
128	7	29	31	72	44
256	8	32	36	90	50

P_T^R = Total pins for DPL using Read-only-Memory DPG.

P_T^{EL} = Total pins for DPL using Encoder for Adder Transfer t_{i-1}^A and 'Local Generation' of t_i^P.

P_T^{NEA} = Total pins for DPL without Encoder for t_{i-1}^A and 'Adjacent Generation' of t_i^P.

P_T^{NEL} = Total pins for DPL without Encoder for t_{i-1}^A and 'Local Generation' of t_i^P.

use of the encoder MATE for 'Adder Transfer' gives reasonable pin count but at the cost of introducing a new cell (full adder) for MATE in the realization of MIAD.

4.4.1.3 Effect of multiplier digit's redundancy on gate and pin complexity of DPL - In the discussion so far, we have assumed that both the multiplier and multiplicand digits are maximally redundant, that is, both can assume values equal to or less than (r-1). However, if the multiplier digit has a redundancy ratio δ lying between 1/2 and 2/3, that is, the maximum magnitude of the multiplier digit is $\leq |2/3\ (r-1)|$, then the multiplier digit can be recoded in Non-Adjacent Format (NAF) [43,49]. In a NAF recoded radix-r multiplier digit, no two adjacent redundant binary digits are nonzero. That is, the recoded multiplier $\hat{x}_i$ is of the form

$$\hat{x}_i = \sum_{j=0}^{k-1} x^*_{i_j} \cdot 2^j \qquad x^*_{i_j} \in \{\bar{1},0,1\}$$

such that $\left|x^*_{i_j}\right| \cdot \left|x^*_{i_{j+1}}\right| \neq 1$ where $\left|x^*_{i_j}\right|$ is the absolute value of the redundant binary digit $x^*_{i_j}$.

With the multiplier digit in NAF format, the number of inputs to all MIRBAs of radix-2^k adder can be reduced to $(\left|\frac{k}{2}\right| + 1)$ from (k+1), by combining two adjacent redundant binary digits of a column into one redundant binary digit. This is shown in Figure 4.29. The reduction in the number of inputs to MIRBAs of the radix-2^k adder causes a corresponding decrease in the gate and pin complexity of the Digit Processing Logic.

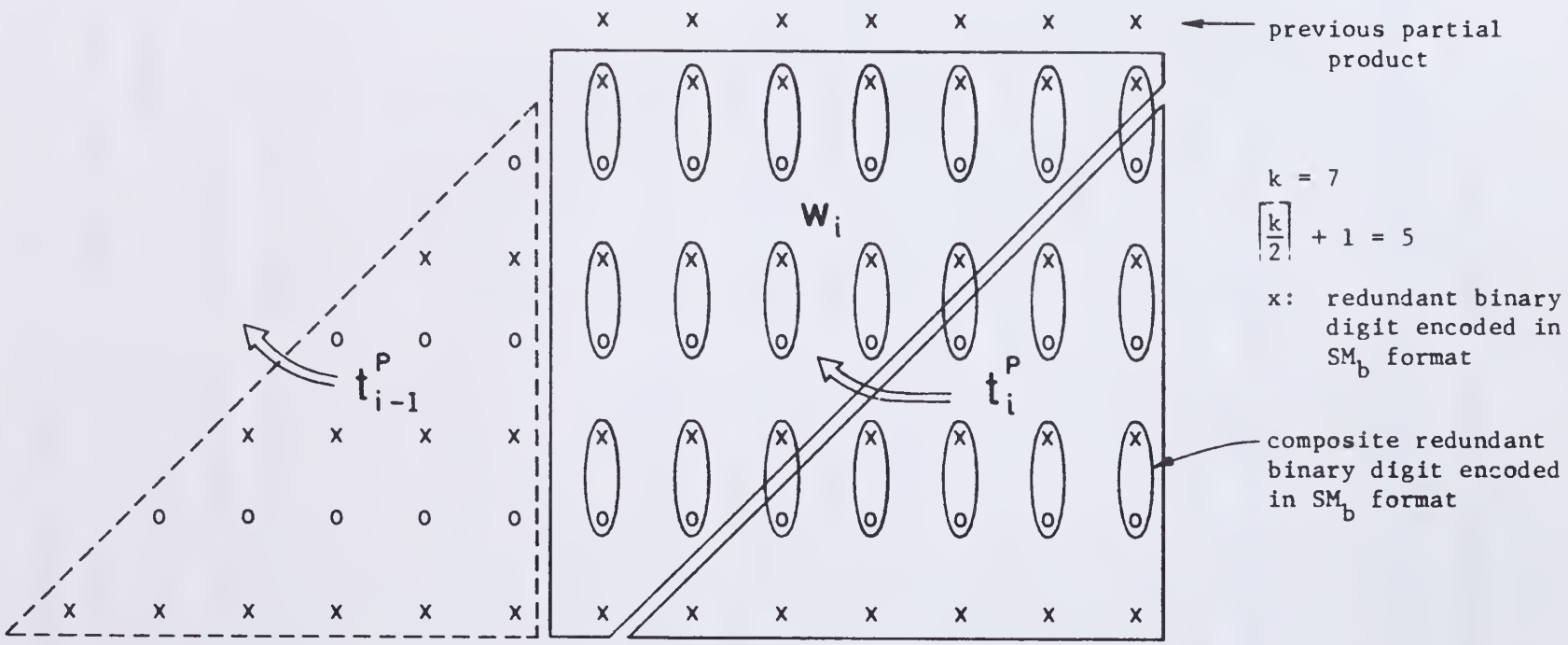


Figure 4.29 Illustration of the Effect of NAF Recoded Multiplier Digit on # of Inputs to MIRBA of Radix-2^k (k = 7) Adder

Gate Complexity

Assuming the sign-magnitude, logic vector encoding LVE_3 for a redundant binary digit, and 'Local Generation' of the product transfer t_i^P, the gates G'_{DPG} for the logic of the Digit Product Generator are given by

G'_{DPG} = gates required for the magnitude bits of the 'product' array w_i and 'transfer' arrays t_i^P + gates required for the generation of sign bits of 'product' and 'transfer' arrays + gates required to combine adjacent bits and their corresponding signs (one of the bits is zero) to form a single (composite) redundant binary digit shown by circles in Figure

= $k^2 + 8k$ + Total # of composite redundant binary digits X gates required to form one composite redundant binary digit.

$$= k^2 + 8k + k\left\lceil\frac{k}{2}\right\rceil \times 4 \qquad (4.29)$$

The above expression shows that the gates required for Digit Product Generator, DPG are increased for $1/2 \leq \delta \leq 2/3$ compared to the maximal redundancy case by an amount equal to $4k\left\lceil\frac{k}{2}\right\rceil + 8(k-1)$.

Further, the complexity of the selector network sADR is also increased because the composite redundant binary digit has to be individually routed to the input of the MIRBAs through the selector sADR.

# of gates needed for the magnitude bits selection = $3k\left\lceil\frac{k}{2}\right\rceil$

# of gates required for sign bits selection = $(2k+1)\left\lceil\frac{k}{2}\right\rceil$

total # of gates required for sADR network = $(2k+1)\left\lceil\frac{k}{2}\right\rceil + 3k\left\lceil\frac{k}{2}\right\rceil$

$$G'_{sADR} = (5k+1)\left\lceil\frac{k}{2}\right\rceil \tag{4.30}$$

From the above it is clear that although the number of gates required for the sign bits' selection is increased compared to the case when $\delta = 1$, the number of gates required for the magnitude bits' selection is almost halved and the overall gates G'_{sADR} required for the selector network sADR is decreased compared to the maximal redundancy case.

However, there is a drastic reduction in the number of gates required for the adder MIAD because of the decrease in the number of inputs to each MIRBA. The gates G'_{MIAD} required are

$$G'_{MIAD} = 26k\left\lceil\frac{k}{2}\right\rceil \tag{4.31}$$

There is no change, due to change in redundancy, in the gates required for either the Digit Sum Encoder DSE or the other remaining selector networks sDSE, sRIB and sTOP. Therefore, the total number of gates G'_{DPL} required for the Digit Processing Logic, when the multiplier digit redundancy is restricted to $1/2 \leq \delta \leq 2/3$ only is

$$G'_{DPL} = G'_{DPG} + G'_{sADR} + G'_{MIAD} + G'_{DSE} + G'_{sDSE} + G'_{sRIB} + G'^{\dagger}_{sROB} + G'_{sTOP}$$

---

†The gates for sROB are calculated on the assumption that we reduce the number of registers in the register file from (k+1) to $\left(\left\lceil\frac{k}{2}\right\rceil + 2\right)$

$$= k^2 + 8k + 4k\left\lceil\frac{k}{2}\right\rceil + (5k+1)\left\lceil\frac{k}{2}\right\rceil + 26k\left\lceil\frac{k}{2}\right\rceil + 16k + 7k +$$

$$4(k+1) + (k+1)\left(\left\lceil\frac{k}{2}\right\rceil + 2\right) + 3(k+2)$$

$$= k^2 + (36k + 2)\left\lceil\frac{k}{2}\right\rceil + 40K + 12 \qquad (4.32)$$

The values of G'_{DPL} and its various components are given in Table 4.4 for different values of the parameter k. A comparison of Table 4.2 and Table 4.4 shows that the reduction in the gates required for digit processing logic, for $1/2 \leq \delta \leq 2/3$,† comes mainly from the drastic reduction in the number of gates necessary for the adder MIAD.

Pin Complexity

Using the same notation as in the case of $\frac{1}{2} \leq \delta \leq 1$, we have

$P_T'^{NEL}$ = Total number of pins required for implementation of DPL, using 'Local Generation' method of t_i^P and no encoder for t_{i-1}^A for $1/2 \leq \delta \leq 2/3$

$$= \text{Max}(k+2,\ 2\left\lceil\frac{k}{2}\right\rceil) + 2\left\lceil\frac{k}{2}\right\rceil + (k+1) + (k+1)$$

$$= (k+2) + 2\left\lceil\frac{k}{2}\right\rceil + (2k+1)$$

$$= 3k + 4 + 2\left\lceil\frac{k}{2}\right\rceil \qquad (4.33)$$

Similarly

$$P_T'^{EL} = \text{Max}(k+2,\ 2\left\lceil\frac{k}{2}\right\rceil) + 2\left\lceil\log_2\left\lceil\frac{k}{2} + 1\right\rceil\right\rceil + (k+1) + (k+1)$$

†Strictly speaking, $\delta = \left\lceil\frac{2}{3}(r-1)\right\rceil \Big/ (r-1)$ which may be slightly larger than 2/3 for certain values of r. In this thesis, however, we shall say that $\delta \leq 2/3$.

Table 4.4 Gate Complexity of DPL Vs Radix for $1/2 \leq \delta \leq 2/3$ and Encoding LVE_3 for a Redundant Binary Digit

Radix $r=2^k$		G'_{DPG}	G'_{MIAD}	G'_{sADR}	G'_{DSE}	G'_{sDSE}	G'_{sRIB}	G'_{sROB}	G'_{sTOP}	G'_{DPL}
r	k									
4	2	28	52	11	32	14	12	9	12	170
8	3	57	156	32	48	21	16	16	15	361
16	4	80	208	42	64	28	20	20	18	480
32	5	125	390	78	80	35	24	30	21	783
64	6	156	468	93	96	42	28	35	24	942
128	7	217	728	144	112	49	32	48	27	1357
256	8	256	832	164	128	56	36	54	30	1556

Table 4.5 Pin Complexity of DPL Vs Radix for $1/2 \leq \delta \leq 2/3$

Radix $r=2^k$		$P'_T{}^{EL}$	$P'_T{}^{NEL}$
r	k		
4	2	12	12
8	3	17	17
16	4	20	20
32	5	23	25
64	6	26	28
128	7	31	33
256	8	34	36

$$= 3k + 4 + 2\left\lceil \log_2 \left\lceil \frac{k}{2} + 1 \right\rceil \right\rceil \qquad (4.34)$$

Values of both the Equations (4.33) and (4.34) are tabulated in Table 4.5.

A comparison of Tables 4.3 and 4.5 shows that by restricting the redundancy ratio to $1/2 \leq \delta \leq 2/3$ for each multiplier digit, one can achieve almost the same number of total pins for DPL, as are achieved by using $\delta = 1$ and encoder MATE, without having to introduce the new cell for MATE. The introduction of MATE destroys the uniformity of the structure of MIAD.

4.4.2 Logic complexity of PE control - The major components of PE control logic are the microinstruction register, MIR, the selector network sMIR, the Zero and Sign Detection Logic ZSD, the microinstruction decoder and control and timing signal generator, TCS. Of these, the gate complexity of only the selector sMIR and ZSD is dependent on the bit width (=k+1) of the PE module because each is one digit wide. The gate complexity of TCS is independent of bit width, if we exclude the file register address decoders from consideration. However, the gate complexity is dependent on the method of implementing the control sequence charts described earlier. The author used the control point technique used in ILLIAC III [45] for the implementation of control sequence charts, in order to calculate the gate complexity of PE control.

4.4.2.1 Gate complexity of PE control - Table 4.6 shows the gates required for each subcontrol of TCS in terms of the number of control points, gates for the control points and the gates required for the conditional generation of control and timing signals.

The last column of the Table 4.6 shows the total number of gates required for each subcontrol. Let G_{TCS} denote the total number of gates required for the Timing and Control Signal Generator, TCS.

$$G_{TCS} \cong 200 \text{ NAND gates}$$

In addition, let G_{DCD}, G_{sMIR} and G_{ZSD} denote the gates required for the logic implementation of the microinstruction decoder, selector sMIR<4:0> and the Zero and Sign Detector. These gates are given by

$$G_{DCD} = 32 \text{ NAND gates}$$

$$G_{sMIR} = 15 \text{ NAND gates}$$

$$G_{ZSD} = 6 \text{ NAND gates}$$

Therefore

$$\begin{aligned} G_{PCL} &= \text{Total \# of gates required by PE control logic excluding storage elements} \\ &= G_{TCS} + G_{DCD} + G_{sMIR} + G_{ZSD} \\ &= 253 \end{aligned} \tag{4.35}$$

4.4.2.2 Pin complexity of PE control - The total number of pins required for the logic implementation of PE local control is the sum of the pins required for microinstruction ports MIP_i and MOP_i and the pins

Table 4.6 Gate Complexity of Various Subcontrols of TCS

Name of subcontrol	# of control points for implementation	# of NAND gates for control points	# of NAND gates for conditional control task signal generation and control memory elements	Total # of NAND gates for subcontrol
R-control	2	13	--	13
T-control	1	6	2	8
F-control	2	12	--	12
DM-control	2	12	22	34
G_{gn}-control	3	21	13	34
G_{ap}-control	2	14	11	25
E-control	7	42	32	74
			Total # of NAND gates = G_{PCL}	200

required for the request-response signals of TCS. Denoting by P_{PCL}, the total pins required by PE local control, we have

$$\begin{aligned} P_{PCL} &= P_{MIP_i} + P_{MOP_i} + P_{TCS} \\ &= 11 + 11 + 14 \\ &= 36 \end{aligned} \tag{4.36}$$

If the multiplier digit has redundancy $1/2 \leq \delta \leq 2/3$, then the number of internal registers in the PE reduces to $\left\lceil \frac{k}{2} \right\rceil + 1$ from (k+1). The number of address bits required to specify the internal register correspondingly reduce to $\left\lceil \log_2 \left(\left\lceil \frac{k}{2} \right\rceil + 1 \right) \right\rceil$. This results in the saving of one pin in the microinstruction ports and thus the pins required for PE control logic reduce to 34.

4.4.3 Overall logic complexity of a PE - The total number of gates, G_{PE}, required for the implementation of a PE is the sum of the gates required for the combinational logic of DPL, the gates required for the PE control logic and the gates required for the implementation of storage registers in the PE. The gates required for DPL and the storage registers are a function of the parameter k (radix-2^k) which represents the bit width of a PE. The gates required for PE control logic are virtually independent of k and are about 250 NANDs. The storage registers in a PE comprise the registers in the register file, buffer registers in DPL and the register MIR in PE control logic. Considering that all the storage registers are made of edge triggered D-type flip-flops, the gates G_{STO} required for the storage registers is given by

G_{STO} = (# of registers in the register file x (k+1) + width of IBR + width of APR + width of GIR + width of MIR) x gates required for one D-type edge triggered flip-flop.

A D-type edge triggered flip-flop requires 6 NAND gates [46]. Therefore, for multiplier digit redundancy ratio $\frac{1}{2} \leq \delta \leq 1$, we have

$$G_{STO} = 6x\left[(k+1)(k+1) + (k+1)+(k+1)+ 2\left\lceil \log_2(k+1)\right\rceil + 11\right]$$

$$= 6\left[k^2 + 4k + 2\left\lceil \log_2(k+1)\right\rceil + 14\right] \qquad (4.37)$$

For multiplier digit redundancy ratio $1/2 \leq \delta \leq 2/3$,

$$G'_{STO} = 6x\left[\left(\left\lceil \frac{k}{2}\right\rceil + 1\right)(k+1) + (k+1)+(k+1)+ 2\left\lceil \frac{k}{2}\right\rceil + 10\right]$$

$$= 6\left[3(k+1) + \left\lceil \frac{k}{2}\right\rceil (2 + k+1) + 10\right]$$

$$= 6\left[3k + (k+3)\left\lceil \frac{k}{2}\right\rceil + 13\right] \qquad (4.38)$$

Now for $\frac{1}{2} \leq \delta \leq 1$

$$G_{PE} = G_{DPL} + G_{PCL} + G_{STO}$$

Substituting the values from Equations (4.24),(4.35) and (4.37)

$$G_{PE} = 37k^2 + 60k + 12\left\lceil \log_2(k+1)\right\rceil + 359 \qquad (4.39)$$

Similarly for $1/2 \leq \delta \leq 2/3$

$$G'_{PE} = G'_{DPL} + G'_{PCL} + G'_{STO}$$

$$= k^2 + k(50 + 42\left\lceil \frac{k}{2}\right\rceil) + 20\left\lceil \frac{k}{2}\right\rceil + 351 \qquad (4.40)$$

The total number of pins required for the logic implementation of the PE is the sum of the pins necessary for the DPL and the pins required for the PE control logic. From the earlier discussion, we find that by restricting multiplier (quotient) digit redundancy to $1/2 \leq \delta \leq 2/3$, a reduction results both in gate complexity and the total number of pins required for the PE. Moreover, the reduction in pins is achieved without destroying the cellularity and structural uniformity of the multi-input adder, MIAD. The total number of pins, P_{PE}, for the PE is given by, for $1/2 \leq \delta \leq 2/3$

$$\begin{aligned} P_{PE} &= (3k + 4 + 2\left\lceil\frac{k}{2}\right\rceil) + 34 \\ &= (3k + 38 + 2\left\lceil\frac{k}{2}\right\rceil) \end{aligned} \tag{4.41}$$

5. INTERACTION WITH MEMORY

5.1 Introduction

The arithmetic structure consisting of Mantissa Processing Logic (MPL) and Exponent Processing Logic (EPL) needs to communicate with Data Main Memory (DMM) for fetching the operands and for storing the results. The major considerations in the design of the interface between the MPL and the DMM are:

a) a PE in the MPL should require minimum possible number of pins for the addresses of the operands, and

b) the interface and the DMM should have high data bandwidth to satisfy the data needs of concurrently processing PEs.

The first point suggests that the DMM address of the operand should not be carried as part of the microinstruction issued by the mantissa control unit, MCU. Instead we should, along with the microinstruction, send in the modifier field either a pointer to an address register in the interface [30] or the address of some location in a small size buffer memory (in the interface) which is preloaded with the operands. In this thesis, the latter approach is adopted.

Since the different PEs in the mantissa processing logic, MPL, are concurrently active and in general operating on digits of different operands, the different PEs may be accessing the interface simultaneously. This requires high bandwidth and leads to a distributed, multiport architecture for the buffer memory. In Section 5.2, the details of the architecture of the mantissa buffer memory called the Local Operand Mantissa Memory (LOMM) are described.

Because the operand address in the LOMM instead of the DMM address of the operand is carried with the microinstruction issued by MCU, a mapping mechanism is necessary in the interface. Besides, the LOMM is of finite size and its contents need to be stored away in the DMM to make space for new operands. A brief description of the mapping mechanism, the loading (storing) of the operands (results) from (to) DMM into (from) LOMM is given in Section 5.3. Finally, Section 5.4 describes some of the factors which determine the word capacity of the buffer memory.

The Local Operand Memory for floating point operands consists of two parts: the Local Operand Mantissa Memory, LOMM and the Local Operand Exponent Memory, LOEM. Figure 5.1 shows the block diagram of Local Operand Memory. The architecture of LOMM and LOEM are independent of each other and depend respectively on the organization and architecture of the Mantissa Processing Logic and the Exponent Processing Logic. In this thesis, we shall be concerned only with the architecture and organization of LOMM and refer to it as the buffer memory. Whenever we speak of the buffer memory operand, the exponent part of the operand is understood to be in the LOEM. The LOEM can be organized in many ways. Since the number of bits required for the operand exponent is not too large, LOEM could consist of simply one memory module of word size equal to the bit width of the exponent and word capacity equal to that of a PEM module.

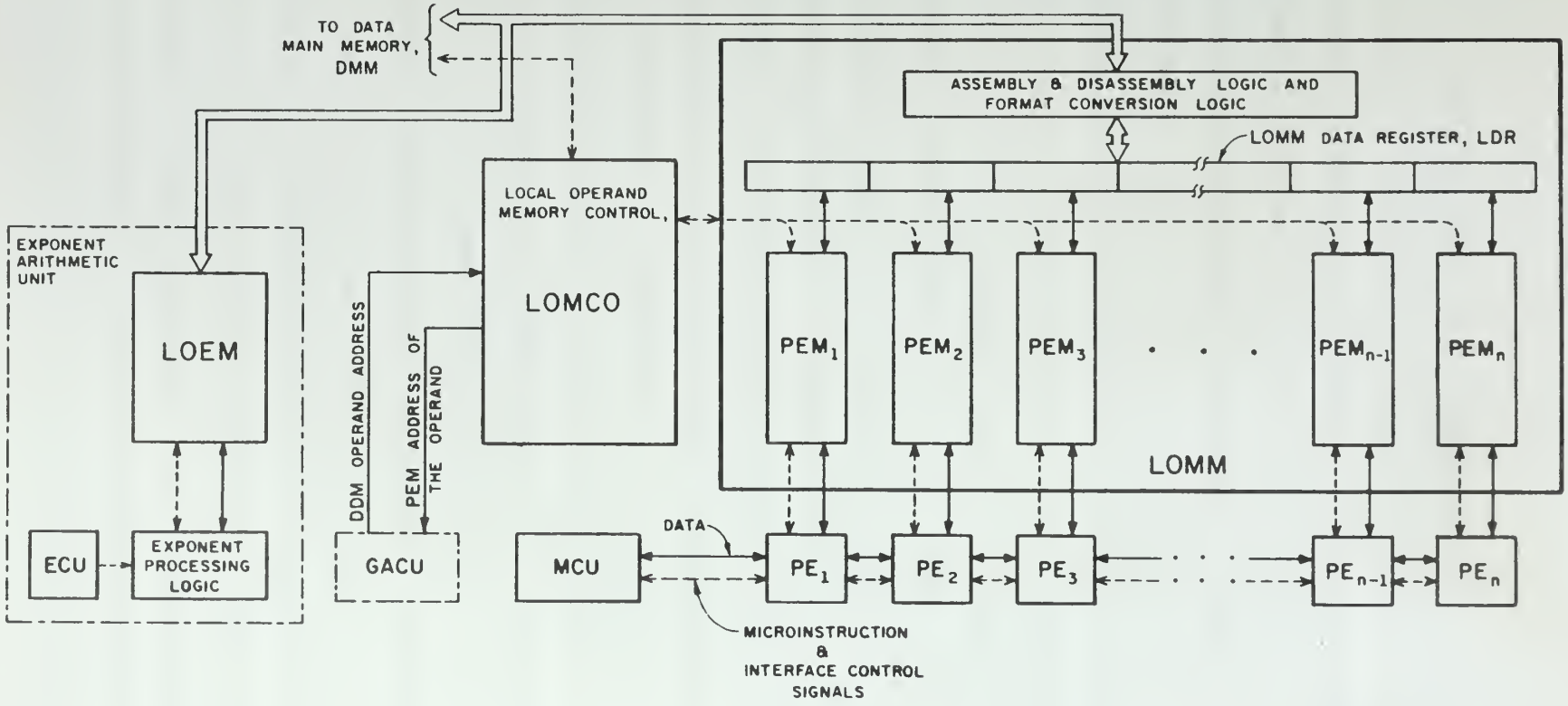


Figure 5.1 Block Schematic Diagram of Local Operand Processor Memory.

5.2 Organization of Local Operand Mantissa Memory, LOMM

The buffer memory, LOMM (Figure 5.1) consists of as many memory modules, called PEMs as there are PEs in the Mantissa Processing Logic. One PEM is associated with one individual PE and has the same bit width (length of an individual PEM word) as the bit width of the PE. Each PEM communicates with its own individual PE and with the LOMM data register, LDR which acts as an interface buffer between the data main memory, DMM and the buffer memory LOMM. An operand is stored across in all the PEs at the same location, each PE carrying one digit of the operand. Each PEM can be considered as a set of digit wide general registers for its corresponding PE. The loading of data from the DMM into PEMs and storing of data from PEMs into DMM is under the control of the Local Operand Memory Control, LOMCO and takes place via buffer data register LDR.

Let us define a Buffer Memory Word as a word formed by concatenating the digits stored across in the same location of all the PEMs of the buffer memory. If the DMM word length is different than the Buffer Memory Word length, the logic for assembly and disassembly of DMM words and Buffer Memory words also form part of the buffer memory logic and is under the control of LOMCO. Further, the operand in interface buffer register LDR is in signed-digit format (each digit carrying the same sign as the sign of the operand) while the DMM word is in conventional sign-magnitude format. The format conversion logic necessary for transformation from sign-magnitude to signed-digit form and vice versa also exists in the data path between DMM and the regiser LDR. Each PEM module of the buffer memory has its own independent access control and

read/write logic. Each PEM is accessed from two sources: its associated PE which fetches/stores data from/into PEM during the execution of microinstructions LPM/SPM; and LOMCO which accesses the individual PEM to read/write a new buffer memory word from/to PEM modules into/from buffer memory data register, LDR. Since the various PEs are accessing their own PEMs not in synchronism but rather independently, LOMCO reads/writes a buffer memory word into/from LDR by requesting access to PEMs individually. Thus, the access between DMM and buffer memory is in parallel whereas the individual PEs access the different digits of the buffer memory word in serial mode.

The distributed multiport architecture for LOMM has the advantages of modularity, easy expandability, high data bandwidth and small number of pins per PE for fetching and storing of operands. Each PEM module in LOMM is a random access monolithic memory. Semiconductor memory chips of N bit capacity are organized as N x 1 words because it leads to minimum number of leads and also this organization makes feasible very effective use of error correcting codes and software diagnostic routines to detect and overcome component failures. Each PEM module can be assembled from these chips--as many as the bit width of the PE. Thus having a separate PEM module associated with each PE provides the necessary flexibility and modularity. Secondly, the small word size of a PEM module permits faster access and cycle times. Multiport architecture permits concurrent operation of the various PEM modules and thus high data rate and data bandwidth. Multiport organization and the communication of each PE with only its own PEM module allows each PEM module

to be of small word capacity and thus reduces the number of pins required for addressing a location in PEM. Each PEM module can be logically considered as an extension of the regiser file of each PE.

5.3 A Description of Buffer Memory Control

The operation of the buffer memory is under the control of LOMCO. The function of LOMCO is two fold:

a) When LOMCO is presented with a DMM address of an operand by the GACU, LOMCO replies back with the PEM address of the operand, if the operand is available in the buffer memory. However, if the operand is not available in the buffer memory, LOMCO searches for an empty location in the buffer memory, fetches the operand from the DMM, loads the operand in the empty location of buffer memory and replies back with the address of the location just loaded.

b) Another function of LOMCO is to store away the contents of those locations in buffer memory which are no longer being used by any of the PEs of the Mantissa Processing Logic to make space for new operands that may be needed by other arithmetic machine instructions.

The LOMCO achieves the above two functions by a table look-up mechanism. There are as many entries in the control table as there are locations in the buffer memory--one entry for each location. Each entry in the control table has five fields as shown in Figure 5.2. The fields labeled DMMA and PEMA respectively denote the DMM operand address and the corresponding PEM address of the operand. The field labeled VCB is a one bit field and denotes the validity control bit for the data stored

PEMA	DMMA	USC	VCB	STF
1	A		0	0
2	D		1	0
3	B		0	1
4	C		0	1
	.	.		
	.	.		
	.	.		
	.	.		
	.	.		
	.	.		
	.	.		
$2^{k+1}-4$	M		1	0
$2^{k+1}-3$	N		1	0
$2^{k+1}-2$	A		1	0
$2^{k+1}-1$	E		1	1
2^{k+1}	F		1	1

Figure 5.2 Structure of Control Table in LOMCO.

in the location specified by PEMA. Whenever an operand is loaded in the buffer memory from the Data Main Memory, DMM, the validity control bit is set to logical '1' state. When a DMM operand address is presented to the LOMCO, an associative table look-up is performed. If there is a match and the field VCB is '1', then the corresponding PEMA entry (i.e., the address of the location in PEM containing the DMM variable (operand)) is sent back as a response. If, however, there is no match or if the validity bit is zero but match takes place,† the LOMCO searches for an empty entry in the table. An empty entry in the table indicates an empty location in the buffer memory. On finding an empty location, LOMCO fetches the operand specified by DMM operand address, stores it in the empty buffer memory location, sets the VCB field to logical state '1' and responds back with PEMA, that is, the address of the operand location in PEM. Another field associated with each entry is the Usage Count or USC field. This field is essential for deciding when to replace the contents of the corresponding buffer memory location. Its function and necessity can be explained in the following way.

In order that all the PEs may be kept usefully busy in processing the microinstructions, it is important that there should be a steady stream of microinstructions being issued to PEs and also that the corresponding operands be available in the buffer memory for processing by the PEs. This implies the use of an instruction look-ahead unit which supplies arithmetic instructions to the GACU of the arithmetic unit.

---

†Such a case can occur when the value of the DMM variable is changed by some other unit accessing the DMM without a corresponding update in the buffer memory.

The LOMCO in cooperation with GACU loads the buffer memory with operands in advance of their processing by the PEs. Since the PEs in the Mantissa Processing Logic operate on the individual digits of an operand in sequence and different PEs may be operating on different operands or on different digits of the same operand, an operand in the buffer memory cannot be replaced by a new operand as long as a PE in MPL is using any digit of the operand. Moreover, if in the arithmetic instruction look-ahead buffer in GACU, there exists an arithmetic instruction which may make use of the operand in buffer memory, the operand should not be replaced by a new operand in order to avoid an unnecessary DMM access by LOMCO. All these control functions are provided through the Usage Count field, USC, in each entry of the table. Contents of the field USC is a tally which is incremented by one every time a match is obtained with the DMMA field in the table entry or an operand is fetched from DMM into buffer memory. This tally in USC is decreased by unity every time a PEM accessing microinstruction exits from the last PE in the MPL. The PEM accessing microinstructions are the LPM and SPM microinstructions as discussed in Section 2.6. When the tally in the usage count field USC goes to zero, it implies that no PE is using the operand in the corresponding buffer memory location and it can be replaced by a new operand from DMM, if necessary.

The final field associated with each entry in the control table is the Store Flag field, STF. It is a one bit field and is set to logical state '1' every time the PEM reference microinstruction SPM exits from the last PE in MPL. Whenever the tally in the USC field goes to zero and

field STF is '1', then the LOMCO would store the contents of all the PEMs at the corresponding location into DMM at a location specified by DMMA field. Note that only the 'final' value of an operand (at the end of a series of calculations involving that operand) is stored in DMM even though there may be a set of store microinstructions, SPMs, in the stream of microinstructions flowing through the PEs. This is because there is no guarantee without tally in USC being zero, that no PE is modifying the contents of its PEM location when a DPM microinstruction referring to the same buffer memory location exits from the last PE in MPL.

5.4 Size of Buffer Memory

The number of words in the buffer memory and hence in each PEM module is dependent, among others, on two main factors.

a) The maximum possible number of PEs which are concurrently accessing different operands in the buffer memory at any time.

b) The ratio of the rate at which the buffer memory can be loaded from DMM and the rate of processing of a microinstruction in a PE.

In order that no PE may be idle due to lack of operands in the buffer memory, it is important that the word capacity of the buffer memory be at least equal to the maximum number of PEs which may at any time be accessing different locations of the buffer memory. The number of PEs concurrently accessing different buffer memory locations in turn depends on the nature of the arithmetic instruction stream, the amount of arithmetic instruction look-ahead in the GACU and the number of PEs in the Mantissa Processing Logic. If there are no data dependencies in

the instruction stream and a constant stream of microinstructions can be issued to the PEs, as many as $\frac{p \times n}{p+1} \simeq n$ PEs may be accessing different operands at any time where n is the number of PEs in the Mantissa Processing Logic and p is the number of operands that can be summed by the Multi Sum microinstruction, MS. Such a case can occur in the evaluation of an arithmetic expression of the form

$$A = \sum_{j=1}^{m} B_j$$

where $B_1, B_2, \ldots, B_m$ are DMM operands.

However, in practical cases there are always data dependencies in the instruction stream and if n is large, there would be some idle PEs and the size of the buffer memory required would always be less than n. From the empirical program studies made by Kuck et al. [47], Knuth [48] and Foster and Riseman [49], we deduce that a buffer memory of word capacity 16 or at the most 32 would be sufficient for all practical purposes.

The word capacity of the buffer memory would also depend on the number of pins available in a PE for addressing its PEM module in the buffer memory.

In our design of the microinstructions LPM and SPM, we have assumed that there are 2^{k+1} words in the buffer memory because (k+1) bits are available in the microinstruction word for addressing the PEM module. In case the word capacity of PEM has to be large, e.g., when k is small, then correspondingly more bits would have to be assigned in the

microinstruction word for the PEM address. For $k \geq 3$, $k+1$ bits are sufficient.

6. IMPLEMENTATION OF MACHINE ARITHMETIC INSTRUCTIONS

6.1 Introduction

In this chapter, we describe how a 'machine' arithmetic instruction can be implemented using the various microinstructions and the particular arithmetic unit organization developed in earlier chapters. The arithmetic unit is organized to process floating point operands. The fractional parts of the operands are processed by the PEs in the Mantissa Processing Logic and the exponent arithmetic is performed by the Exponent Processing Logic. Since the exponent arithmetic involves simply taking the sum or difference of the exponents of two operands, no details are given of the method of processing the exponents. Instead, the major emphasis is put on the sequence of microinstructions which are issued by the Mantissa Control Unit, MCU, to process the 'machine' arithmetic instruction.

6.2 Implementation of 'Machine' Arithmetic Instructions

6.2.1 Global description of the processing of a 'machine' arithmetic instruction - When the instruction look-ahead unit in the machine, of which this Arithmetic Unit forms a part, detects an arithmetic instruction, it sends the arithmetic instruction to the Arithmetic Unit for processing. The GACU part of the local arithmetic control acts as the interface between the arithmetic unit and the instruction look-ahead unit. On receiving the machine instruction, the GACU calls upon the buffer memory control, LOMCO to provide the GACU with the buffer memory

address of the data operand (referred in machine arithmetic instruction). (If the data operand is not present in the buffer memory, LOMCO will fetch the operand and then provide the buffer memory address as explained earlier in Chapter 5.) The GACU now sends the machine instruction, with the DMM operand address replaced by the corresponding buffer memory address, to the MCU. MCU decodes the arithmetic instruction and calls upon the exponent control unit, if necessary, to calculate the sum/difference of the exponents of the operands involved. The microinstructions for the processing of mantissas are then issued either concurrently with exponent processing or on response from the exponent control unit depending on the arithmetic instruction. (In the case of an Add or Subtract instruction, the difference of the exponents must be available for operand alignment before mantissa processing can begin.) The sequence of microinstructions issued by the MCU for the mantissa processing depends on the OP-code (for example, Add, Subtract, Multiply, etc.) of the arithmetic instruction and whether the instruction is single address, double address, or the three address type. In the following discussion, a single address type of 'machine' arithmetic instruction is assumed. That is, each instruction is of the format OP-code , EA where OP-code field carries the mnemonic and EA is the effective DMM address of the operand. The other operand, if necessary, is implicitly assumed to be in the Accumulator which is distributed in the PEs of the Mantissa Processing Logic. The Register INR1 of the register file acts as the Accumulator register in each PE.

MCU also monitors the development of any exceptional conditions like exponent overflow/underflow, etc. during arithmetic instruction processing and reports such an occurrence to the GACU. GACU in turn may pass this status information to other parts of the machine; e.g., instruction look-ahead and fetch unit for branching and other decisions. Due to the most-significant-digit-first nature of arithmetic processing, the occurrence of singular conditions can be detected by MCU before the execution of arithmetic instruction is fully complete.

6.2.2 Floating point Addition - Let the Floating Point ADD instruction be given by FPA , EA where EA is the effective DMM address of the Addend operand. Let ea be the corresponding buffer memory address of the Addend. Once the buffer memory address ea is known, the MCU calls upon the Exponent Control Unit, ECU, to calculate the difference of exponents of the operands in Accumulator and ea, and issues the microinstructions for the right shift of the appropriate operand to align the operands, the summation of the operands and finally checks for mantissa overflow and takes the necessary corrective action. This could be followed by normalization, assimilation to conventional form and then storing of the result operand.

6.2.2.1 Mantissa processing microprogram - Assuming, without loss of generality, that the Addend is to be shifted for operand alignment, the sequence of microinstructions issued by MCU is shown below.

Microinstruction				comments
LPM	2	ea	;	INR2 ← PEM[ea]
RS	2	0	;	As many right shift micro-
RS	2	0	;	instructions are issued as
	.			is the difference in ex-
	.			ponents of the Addend and
RS	2	0	;	Augend.
SS			;	INR1 ← INR1 + INR2

The sequence of microinstructions given above loads the addend in register INR2, aligns the operands and then sums the operands. The sequence of microinstructions for normalization and assimilation of operands is discussed in Sections 6.2.6 and 6.2.7 respectively.

6.2.2.2 Mantissa overflow correction - In the d-vector representation of the sum S

$$S \equiv s_0 \cdot s_1 \, s_2 \, s_3 \cdots s_n$$

it is possible that $|s_0| = 1$. However, this is only an indication of potential overflow. A mantissa overflow occurs only when $s_0 \cdot s_i > 0$ where s_i is the first most significant non-zero digit in the d-vector representation of S. Due to the use of an RBA-2 using the sign-magnitude logic vector encoding LVE_3 for the redundant binary digit, in the implementation of microinstruction SS, bogus overflow [35] would occur quite often. The bogus overflow occurs whenever $s_0 \cdot s_i < 0$ because the sum can always be recoded such that $s_0 = 0$. For example, the sum $1.0\bar{3}2\bar{1}$ can always be recoded into its algebraic equivalent $0.972\bar{1}$.

The mantissa overflow can be corrected by shifting the sum right by one digital position and correspondingly adjusting the exponent of the

sum. In the case of bogus overflow, however, this procedure would cause a loss of one significant digit (k bits for radix-2^k arithmetic) unnecessarily. However, this can be taken care of during normalization of the sum if the shifted-out digit of the sum is saved in the End Unit and reintroduced during left-shifting of the operand. The left shift of the sum after normalization recoding is done to eliminate the leading zeros in the recoded sum.

6.2.3 Floating point Subtraction - The processing for Floating Point Subraction is exactly identical to that for Floating Point Addition except that in the Mantissa Processing Microprogram, the microinstruction SS is preceded by the microinstruction TI 2,2. This microinstruction reverses the sign of each digit of the subtrahend. The sequence of microinstructions for mantissa processing is as follows.

Microinstructions				comments
LPM	2	ea	;	INR2 ← PEM[ea]
RS	2	0	;	
RS	2	0	;	
	⋮			operand alignment
RS	2	0	;	
TI	2	2	;	INR2 ← (-1) . INR2
SS			;	INR1 ← INR1 + INR2

6.2.4 Floating point Multiplication - Let the Floating Point Multiply instruction be denoted as FPM , EA where EA is the effective DMM address of the Multiplier operand. The operand in the accumulator is the implicitly assumed multiplicand operand. If ea is the corresponding buffer memory address of the multiplier operand, the processing for

Floating Point Multiply instructions involves the following steps by MCU. MCU calls upon the Exponent Control Unit to sum the exponents of the operands in the accumulator and at LOEM address specified by ea; concurrently, the MCU issues the sequence of microinstructions to PEs to form the double length product in the PEs and finally to check for any exceptional conditions. Mantissa overflow cannot occur because the mantissas of both the operands are less than unity. However, exponent overflow may take place.

6.2.4.1 Microprogram for mantissa processing - The mantissa processing for the multiplication instruction involves the generation of partial products and the final product digits. For processing, the multiplier and multiplicand operands are respectively in file registers INR3 and INR2 whereas the Accumulator register INR1 is used to form and accumulate the partial products. Unlike the conventional multipliers, the most significant half of the final double length product is in the Multiplier register INR3 and the least significant half is in the Accumulator register INR1.

Because the partial products are formed beginning with the most significant digits, the most significant digits of the product are formed first. Also, to achieve maximum precision the partial product is shifted left during each step instead of the multiplicand being shifted right. Due to the left shift of the partial product, two problems immediately arise.

a) During the left shift of the Accumulator register (partial product), not one but two digits (the digits to the left and right of

the radix point) are shifted out. These two digits need to be recoded into a final product digit to be stored into the multiplier register and a residual digit to be added to the next partial product in the next step. Pisterzi [30] has shown that a recoder with r^2 states is necessary for this purpose. Such a recoder can be conceptually looked upon as an extension of the adder network to the left. However, in our case, due to the existence of bogus overflow in RBA-2, the basic cell of the adder MIRBA, the recoder's logic design would have to be different.

b) Another problem due to left shifting of the partial product is that the digits of the most significant half of the final product which become available one by one as the output of the recoder (connected to the most significant digital position of the adder) need to be stored in the multiplier register and/or the buffer memory. In order that these product digits may be stored in proper order in the multiplier register, the product digit (output of recoder) needs to be stored in the least significant digital position of the multiplier register because this position is vacant due to the left shift of the multiplier register. But MCU communicates with and knows the state of only the most significant PE. The solution to this problem is to send the value of the digit to be placed in the least significant digital position with the left shift multiplier microinstruction. As a matter of fact. the particular definition of the left shift microinstruction was contrived to serve specially this purpose only.

The MCU forms the partial products by issuing a sequence of a set of three microinstructions as many times as the number of digits in the

multiplier operand. The three microinstructions are 'Left Shift Multiplier' to examine the multiplier digit, 'Form Multiple and Add' to form the partial product and 'Left Shift Accumulator' to shift the partial product. The microprogram for the formation of the double length product for six digit long multiplier and multiplicand operands is given below. m_i and P_j (i=1,2,...,6 and j=1,2,...,11,12) respectively denote multiplier and Product digits.

Microinstruction				comments
LPM	3	ea	;	INR3 ← PEM[ea] (≡ Multiplier)
TD	1	2	;	INR2 ← INR1 (≡ Multiplicand)
LDC	1	0	;	INR1 ← 0
LS	3	0	;	MCU ← m_1, INR3 ← r.INR3
FMA	m_1		;	INR1 ← INR1 + INR2 . m_1
LS	1	0	;	MCU ← P_1 , INR2 ← r.INR2
LS	3	P_1	;	MCU ← m_2,INR3$_6$ ← P_1, INR3 ← r.INR3
FMA	m_2		;	These achieve the partial products for the rest of the multiplier digits m_2, m_3,..., m_6 in the same way as the sequence of immediately preceding four microinstructions
LS	1	0	;	
LS	3	P_2	;	
FMA	m_3		;	
LS	1	0	;	
LS	3	P_3	;	
FMA	m_4		;	
LS	1	0	;	
LS	3	P_4	;	
FMA	m_5		;	
LS	1	0	;	
LS	3	P_5	;	
FMA	m_6		;	
LS	1	0	;	
LS	3	P_6	;	

A pictorial representation of the flow and execution of the above sequence in a 6 PE Mantissa Processing Logic is shown in Figure 6.1. In

MICROINSTRUCTION REGISTER					
MIR1	MIR2	MIR3	MIR4	MIR5	MIR6
LPM 3,ea					
TD 1,2	LPM 3,ea				
LDC 1,0	TD 1,2	LPM 3,ea			
LS 3,0	LDC 1,0	TD 1,2	LPM 3,ea		
	LS 3,0	LDC 1,0	TD 1,2	LPM 3,ea	
FMA,m_1		LS 3,0	LDC 1,0	TD 1,2	LPM 3,ea
	FMA,m_1		LS 3,0	LDC 1,0	TD 1,2
		FMA,m_1		LS 3,0	LDC 1,0
			FMA,m_1		LS 3,0
LS 1,0				FMA,m_1	
	LS 1,0				FMA,m_1
LS 3,P_1		LS 1,0			
	LS 3,P_1		LS 1,0		
FMA,m_2		LS 3,P_1		LS 1,0	
	FMA,m_2		LS 3,P_1		LS 1,0
		FMA,m_2		LS 3,P_1	
			FMA,m_2		LS 3,P_1
LS 1,0				FMA,m_2	
	LS 1,0				FMA,m_2
LS 3,P_2		LS 1,0			
	LS 3,P_2		LS 1,0		
FMA,m_3		LS 3,P_2		LS 1,0	
	FMA,m_3		LS 3,P_2		LS 1,0
		FMA,m_3		LS 3,P_2	
			FMA,m_3		LS 3,P_2
LS 1,0				FMA,m_3	
	LS 1,0				FMA,m_3
.	.	LS 1,0			
.	.	.	LS 1,0		
.	.	.	.	LS 1,0	
LS 3,P_5		.	.	.	LS 1,0
	LS 3,P_5		.	.	.
FMA,m_6		LS 3,P_5		.	.
	FMA,m_6		LS 3,P_5		.
		FMA,m_6		LS 3,P_5	
			FMA,m_6		LS 3,P_5
LS 1,0				FMA,m_6	
	LS 1,0				FMA,m_6
LS 3,P_6		LS 1,0			
	LS 3,P_6		LS 1,0		
---		LS 3,P_6		LS 1,0	
---	---		LS 3,P_6		LS 1,0
---	---	---		LS 3,P_6	
---	---	---	---		LS 3,P_6
---	---	---	---	---	

MULTIPLIER REGISTER, INR3					
$INR3_1$	$INR3_2$	$INR3_3$	$INR3_4$	$INR3_5$	$INR3_6$
m_1					
	m_2				
		m_3			
			m_4		
m_2				m_5	
	m_3				m_6
		m_4			
			m_5		
m_3				m_6	
	m_4				
		m_5			
			m_6		
m_4				0	
.	m_5				
.	.	m_6			
m_6	.	.	0		
	0	.	.	P_1	
		P_1	.	.	P_2
			P_2	.	.
				P_3	.
					P_4
0					
	P_1				
		P_2			
			P_3		
P_1				P_4	
	P_2				P_5
		P_3			
			P_4		
				P_5	
					P_6

REGISTER IN MCU
m_1
0
P_1
m_2
0
P_2
m_3
P_3
.
P_5
m_6
P_6
0

ACCUMULATOR REGISTER, INR1					
$INR1_1$	$INR1_2$	$INR1_3$	$INR1_4$	$INR1_5$	$INR1_6$
0					
	0				
		0			
			0		
				0	
1^P1					0
	1^P2				
1^P2		1^P3			
	1^P3		1^P4		
		1^P4		1^P5	
			1^P5		1^P6
				1^P6	
2^P1					0
	2^P2				
2^P2		2^P3			
	2^P3		2^P4		
		2^P4		2^P5	
			2^P5		2^P6
				2^P6	
3^P1					0
	3^P2				
3^P2		3^P3			
.	3^P3		3^P4		
.	.	3^P4		3^P5	
5^P2	.	.	3^P5		3^P6
	5^P3	.	.	3^P6	
		5^P4	.	.	0
			5^P5	.	.
				5^P6	.
6^P1					0
	6^P2				
6^P2		6^P3			
P_7	6^P3		6^P4		
	P_8	6^P4		6^P5	
		P_9	6^P5		6^P6
			P_{10}	6^P6	
				P_{11}	0

Figure 6.1 A Pictorial Representation of the Flow of the Sequence of Microinstructions for Mantissa Processing Logic for Multiplication.

this figure,

$_iP_j$ is the j^{th} digit of the portion of the i^{th} accumulated partial product which is in the Accumulator register and

P_j is the j^{th} digit of the final product and is

$$P_j = {}_jP_1 \qquad 1 \leq j \leq 6$$

$$P_j = {}_6P_{j-5} \qquad 7 \leq j \leq 11$$

$$P_{12} = 0$$

The column labeled 'Register in MCU' is a one digit wide register which holds the multiplier digit when the multiplier register is shifted left for examining and selecting the next multiplier digit for partial product formation. This register is also used to hold the product digit, from the output of the accumulator overflow recoder, for storing in the least significant digital position of the multiplier register via the Left Shift Accumulator (LS 3,P_i) microinstruction.

If the multiplier digit m_j' in the microinstruction FMA,m_j' is to have a redundancy ratio $\delta \leq 2/3$, then the two consecutive digits m_j and m_{j+1} of the multiplier register have to be examined to generate one modified multiplier digit m_j'. The algebraic design of such a redundancy recoder is given in the Appendix A-1. The sequence of microinstructions for mantissa processign when the multiplier digit redundancy is $\delta \leq 2/3$ would remain the same as before except that the fourth microinstruction LS 3,0 will be immediately followed by another LS 3,0 microinstruction in order to bring m_2 to the recoder. For the rest of the modified multiplier digits m_j' (j > 1), digit m_{j+1} only needs to be brought into

MCU because m_j is already known from the previous step. Note that there may be one more modified multiplier digit than the number of multiplier digits in the original multiplier operand to maintain algebraic equivalence in the two forms of the same multiplier operand.

6.2.5 Floating point Division - The processing for Floating Point Division is almost identical to that for Multiplication except that the quotient digits must be determined by examination of the partial remainder. Division is performed by repetitive additions and shifts. In the Floating Point Divide instruction FPD , EA, the effective DMM address of the divisor operand is given by EA and the Dividend is implicitly assumed to be the operand in the Accumulator. The processing by MCU for Floating Point Divide involves calling upon the Exponent Control Unit to take the difference of the exponents of the dividend (accumulator) and the divisor at the buffer memory address ea, and processing of the mantissa to calculate the quotient digits. The exceptional conditions are the possible exponent underflow and a zero value of the divisor.

6.2.5.1 Microprogram for mantissa processing - The major problems in the implementation of Division in the arithmetic unit under consideration are:

a) the storage of double precision dividend,

b) the calculation of the quotient digits,

c) the placement of quotient digits in the PEs, and

d) the extension to the left of the Accumulator and the Adder Network to take care of a shifted partial remainder.

We would not discuss the above problems in detail except indicating the possible solutions. For details, the reader should refer to Pisterzi [30]. The double precision dividend is stored in two registers--the Accumulator register INR1 and the multiplier register INR3. The Accumulator register INR1 holds the most significant half of the dividend and INR3 holds the least significant half. At the end of the processing, they respectively hold the remainder and the quotient. Register INR2 will hold the divisor.

Because of the redundant number representation for the quotient digit, the quotient digit can be calculated by a 'model division' [50] which uses only truncated version of the divisor and shifted partial remainders. It is shown in the Appendix A-2 that for radix 2^k ($k \geq 3$), 3 digits of the divisor and 2 digits of the fractional part in addition to the integer part of the shifted partial remainder are sufficient for the calculation of the quotient digit with redundancy ratio of 2/3 or 1. However, for radix-4, one more digit each of the divisor and partial remainder are necessary if the quotient digit has redundancy ratio of 2/3. But for maximally redundant quotient digits, we use the same number of digits as for $k \geq 3$. The examination of the operand digits for quotient calculation in the MCU is done by shifting the operands left as many times as the number of digits necessary. Examination of the divisor digits needs to be done only once at the beginning since the same digits take part in the calculation at every step of a quotient digit. However, since the partial remainder changes, it has to be shifted every time. But since the unshifted divisor and the radix-r

shifted partial remainder are necessary in the PEs for calculation of a new partial remainder, the examination of the operand digits is done by shifting another register which contains a copy of the operand whose digits are to be examined. File register INR4 can be used for that purpose.

The quotient digit is stored in the vacant least significant digital position of the register INR3 by using the Left Shift microinstruction just as in the case of Multiplication. The quotient digit is sent in the modifier field of the Left Shift microinstruction issued by the MCU.

Because of the characteristics of the Division process, the shifted partial remainder in the accumulator would always be less than r in absolute magnitude. Thus the overflow recoder that was used in the Multiplication process can be used to store the integer part of the shifted partial remainder.

Note that the technique used for the 'Model' Division is completely independent of the architecture of our Arithmetic Unit. It can be done by Table look-up or by any other method depending on the time and cost considerations. Pure table look-up is too expensive for any reasonable radix greater than 4. We propose that the quotient digit be calculated in MCU serially one bit at a time for radix $\geq$ 8 and then assembled into a radix-r digit before calculating the next partial remainder.

The sequence of microinstructions for Floating Point Division is very similar to that for Multiplication and hence would not be given here.

6.2.6 Normalization of operands - An operand is considered normalized if it satisfies the definition 3, given in Section 3.5.1, of a normalized number. The major steps in the normalization process are left shifting of the signed-digit operand till there are no leading zeros, recoding the shifted operand by the 'Normalize Recode' microinstruction, NR and finally left shifting the recoded operand to remove the leading zeros, if any were created by the microinstruction NR.

Because of the interface control signal Z_1 between PE_1 and the MCU, there is no need to launch a Left Shift microinstruction to examine the leading digit for zero magnitude. Simply monitoring of Z_1 is sufficient and this has the advantage that no overshift of the operand would take place during normalization process.

Note that since the microinstruction NR operates only on the operand in the Accumulator register INR1, the operand should be placed in INR1 for the normalization process.

6.2.7 Assimilation of signed-digit operand - The process of Assimilation converts the signed-digit operand whose different digits carry, in general, different signs to a form in which each digit has the same sign. This sign is the sign of the operand. The procedure and the sequence of microinstructions necessary for Assimilation is identical to that for Normalization except that the microinstruction AR instead of NR is launched for recoding the operand.

7. SUMMARY AND CONCLUSIONS

7.1 Summary and Discussion of Results

Chapter 1 described the characteristics and the constraints of the newly emerging technology of Large Scale Integration (LSI) and its implications for the design of a digital system. Based on this discussion, a set of desirable characteristics for an Arithmetic Unit were formulated and a limited interconnection arithmetic unit as proposed by Pisterzi [30] was chosen as a vehicle to study the arithmetic and logic design aspects of the basic module of such an arithmetic unit.

Chatper 2 described briefly the logical organization and mode of operation of the arithmetic unit--especially of the Mantissa Processing Logic (MPL). The MPL is composed of a linear cascade of identical logic modules called Processing Elements (PEs) which execute a sequence of microinstructions issued to MPL by the Mantissa Control Unit (MCU). The MCU is an interpreter for the 'machine' arithmetic instructions like 'Multiply', 'Add', etc., and issues a sequence of microinstructions for processing. The salient feature of the processing in MPL is that a microinstruction issued by the MCU is not broadcasted to all the PEs in the MPL. Instead, the microinstruction is executed by the PEs in sequence starting with the most significant PE. (The 'significance' of a PE is the same as the arithmetic significance of the operand digit contained in that PE.) The method of processing in the MPL was illustrated by an example which showed how the various microinstructions in the microinstruction stream could be pipelined. This pipelining

feature allows the meshing in of machine arithmetic instructions even before all the result digits of a previous machine instruction have been calculated. The discussion in this chapter forms the framework for the material in the subsequent chapters.

Chapter 3 is concerned with the arithmetic design of the Processing Element. Due to the digit serial nature of the arithmetic processing and the desirability of limited intercommunication between PEs, redundant number system is a necessity. The number system was chosen to be Signed Digit and maximally redundant firstly because the conversion from the conventional number representation of sign and magnitude to the maximally redundant and vice versa is very simple, and secondly, the radix-2^k arithmetic can be realized in terms of identical stages of redundant binary $\{\bar{1},0,1\}$ arithmetic structures. This gives the required repetitive and uniform logical structure to the internal logic of the PE. Then a set of simple arithmetic microinstructions sufficient for the implementation of four basic arithmetic operations are defined and their digit algorithms are described by their arithmetic transfer functions and their algebraic implementation. The particular algebraic implementation of the digit algorithm is influenced by LSI technology constraints of regularity of logic structure, simplicity of the basic cell of the logic structure and the least number of pins for the module. The regularity of logical structure is obtained by implementing the radix-2^k multi-input adder as a linear cascade of k multi-input redundant binary adder. Each multi-input redundant binary adder in turn is implemented as a tree structure of still simpler 2 inputs or 3 inputs redundant binary adders. A definition

of normalized operands was developed and its influence on the arithmetic properties of overall processing and the complexity of quotient digit calculation was also discussed. The definition for normalized operands chosen was such that processes of 'normalization' and 'assimilation' could share the same logic.

Chapter 4, which is the major contribution of this thesis, treats the logic design of the Processing Element. In this chapter, the gate complexity and pin complexity of the Processing Element are shown to be related to the bit width of the Processing Element (radix of arithmetic processing in MPL) and the redundancy in the multiplier/quotient digit used to form the partial products in the process of multiplication. The major components of the Processing Element are the Register File for the storage of active operands, the Digit Processing Logic which is essentially a combinational logic network for the data transformation, and the Processing Element Control which receives and decodes the microinstruction and generates the necessary sequence of control signals to condition the combinational network DPL. The number of gates and pins required for the DPL are very strongly dependent on the bit width of the Processing Element whereas the number of gates and pins required for the PE control is almost independent of the bit width of the module. From the inspection of Tables 4.3 and 4.5, it is clear that 'local generation' of collective Product Transfer t_i^P should be used to keep down the number of pins necessary on the PE module.

An examination of Tables 4.2 and 4.4, which give respectively the number of gates required in the implementation of DPL for multiplier

digit's redundancy ratio of 1 and 2/3, leads to the conclusion that redundancy ratio of 2/3 should be employed for the multiplier and quotient digit. This would require the existence of a multiplier digit recoder in MCU because the digits of the multiplier operand in the MPL have redundancy ratio of unity. But the multiplier digit recoder is very simple. A still further advantage of restricting the redundancy ratio of the multiplier/quotient digit to $\leq 2/3$ is that α_{FMA}--the number of PEs which must cooperate with a given PE in the execution of microinstruction FMA--would always remain 2 irrespective of the radix of the multiplier digit, when the MIRBAs in MIAD are implemented as a log-sum tree of RBA-2s only. This can be seen from the Table 7.1.

Table 7.1 Values of α^b and α_j when the multiplier/quotient digit redundancy ratio is $1/2 \leq \delta \leq 2/3$

Radix $r = 2^k$		# of inputs to a MIRBA $I = \lceil \frac{k}{2} \rceil + 1$	$\alpha^b = 2\lceil \log_2 I \rceil$	$\alpha_j = \left\lfloor \frac{\alpha^b - 1}{k} \right\rfloor + 1$
r	k			
4	2	2	2	2
8	3	3	4	2
16	4	3	4	2
32	5	4	4	2
64	6	4	4	2
128	7	5	6	2
256	8	5	6	2

Finally, inspection of Table 4.5 shows that the DPL requires only 36 pins for radix-256, that is, for a 8 bit width of the PE module. Since the PE control requires 36 pins also, an eight bit wide PE module should be employed in the Mantissa Processing Logic in order to balance the arithmetic processing cost in DPL and PE control cost. This requires a total of 72 pins on the PE module package and which is by no means unreasonable by the standards of today's technology.

A negative aspect, from the LSI viewpoint, of the structure of Mantissa Processing Logic should be noted here. Since the microinstruction flow from one PE to the other instead of being broadcast from the MCU, the number of pins required for the PE control are doubled in the present structure. Moreover, the request-response strategy of PE coordination control also doubles the number of pins required compared to a synchronous control synchronized to a central clock. However, the asynchronous control has the advantage that any number of PEs can be concatenated together more easily to achieve any desired precision without worrying about the clock skew problems. It should be noted, however, that the arithmetic and logic design of the DPL as described in this thesis is independent of the nature of PE control and the same DPL design can be used to design a PE module for a bus-structured and synchronous Mantissa Processing Logic.

In Chapter 5, a brief description was given of the logic organization and structure of a buffer memory which acts as an interface between the arithmetic unit and the Data Main Memory. The major characteristic of the buffer memory is that communication between the buffer memory and

Data Main Memory is on word level whereas the communication between the buffer memory and Mantissa Processing Logic is on a digit serial basis. It was further argued that the size of the buffer memory in words is fairly small--of the order of 16 to 32 words.

Chapter 6 showed how various machine arithmetic instructions could be implemented using the microinstructions.

7.2 Suggestions for Further Work

Reliability and availability considerations were not addressed in this thesis. Since microinstructions flow from any PE to its adjacent PE, it is important that all the consecutive PEs operate properly in order for the Arithmetic Unit to operate properly. Determining organizational modifications in the interconnection structure of the PEs which would facilitate the automatic reconfiguration of properly operating PEs to yield a working Arithmetic Unit with degraded performance, in the presence of faulty PEs, is a very important area of further investigation.

Because the processing in the Arithmetic Unit takes place on a digit-by-digit basis starting with the most significant digit, this Arithmetic Unit structure has a potential for implementing a dynamically varying precision arithmetic. But due to the possibility of different PEs working concurrently on digits of different operands, certain structural modifications would be necessary. Investigation of such modifications is another interesting area of investigation. One possible solution may be the use of some kind of 'end-of-the-word' marker as the delimiter

for the precision of the operands and the use of a bus-structure to inform the MCU when the last digits of the operands have been operated on.

A simulation of the Arithmetic Unit using data from real programs would be interesting and useful to determine the useful word capacity of a PEM module.

Finally, the logic design of the MCU and the GACU should be performed to determine the actual gate complexity of this module.

LIST OF REFERENCES

[1] Berg, R. O. and Jack, L. A., "System and Logic Design for the Effective Use of LSI," Proceedings of the National Telecommunications Conference, Atlanta, Ga., Nov. 1973, pp.

[2] Smith, M. G., "LSI and Systems Architecture in the 1970's," First USA-JAPAN Computer Conference, 1972, pp. 182-192.

[3] Conway, M. E. and Spandorfer, L. M., "A Computer System Designer's View of Large Scale Integration," 1968 Fall Joint Computer Conference, AFIPS Proc., Washington, D.C.: Spartan 1968, pp. 835-845.

[4] Jennings, R. C., "Design and Fabrication of a General Purpose Airborne Computer Using LSI Arrays," Digest of IEEE Computer Group Conference, June 1968, pp. 50-54.

[5] Beuscher, H. J. and Toy, W. N., "Check Schemes for Integrated Microprogrammed Control and Data Transfer Circuitry," IEEE Trans. EC, Vol. C-19, No. 12, Dec. 1970, pp. 1153-1159.

[6] Beelitz, H. R., Levy, S. Y., Linhardt, R. J., and Miller, H. S., "System Architecture for Large-Scale Integration," 1967 Fall Joint Computer Conference, AFIPS Proc., Washington, D.C.: Spartan 1967, pp. 185-200.

[7] Clark, W. A., "Macromodular Computer Systems," 1967 Spring Joint Computer Conference, AFIPS Proc., Washington, D.C.: Spartan 1967, pp. 337-401.

[8] Podraza, G. V., Gregg, R. S., Jr., and Slager, J. R., "Efficient MSI Partitioning for a Digital Computer," IEEE Trans. EC, Vol. C-19, No. 11, Nov. 1970, pp. 1020-1028.

[9] Cserhalmi, N., Lowenschuss, O., and Scheff, B., "Efficient Partitioning for the Batch-fabricated Fourth Generation Computer," 1968 Fall Joint Computer Conference, AFIPS Proc., Washington, D.C.: Spartan 1968, pp. 857-865.

[10] Chen, T. C., "Overlap and Pipeline Processing" in Introduction to Computer Architecture, Edited by H. S. Stone, Chicago, Science Research Associates, Inc., 1975, pp. 375-431.

List of References (continued)

[11] Ramamoorthy, C. V. and Economides, S. C., "Fast Multiplication Cellular Arrays for LSI Implementation," 1969 Fall Joint Computer Conference, AFIPS Proc., Washington, D.C.: Spartan 1969, pp. 89-98.

[12] Gex, A., "Multiplier-Divider Cellular Array," Electronics Letters, 29th of July 1971, Vol. 7, No. 15, pp. 442-444.

[13] Kingbury, N. G., "High Speed Binary Multiplier," Electronics Letters, 20th of May 1971, Vol. 7, No. 10, pp. 277-278.

[14] Wallace, C. S., "A Suggestion for a Fast Multiplier," IEEE Trans. EC, Feb. 1964, pp. 14-17.

[15] Majithia, J. C. and Kitai, R., "An Iterative Array for Multiplication of Signed Binary Numbers," IEEE Trans. EC., Vol. C-20, No. 2, Feb. 1971, pp. 214-216.

[16] Baugh, C. R. and Wooley, B. A., "A Two's Complement Parallel Array Multiplication Algorithm," IEEE Transactions on Computers, Vol. C-22, No. 12, pp. 1045-1047.

[17] Majithia, J. C., "Non-Restoring Binary Division Using a Cellular Array," Electronics Letters, June 1970, pp. 303-304.

[18] Majithia, J. C. and Kitai, R., "Fast Multiplier/Divider Array Using a Controlled Iterative Array," private communication.

[19] Bjorner, Dines., "A Flow-Mode, Self-Steering, Cellular Multiplier-Summation Processor," BIT 10, 1970, pp. 125-144.

[20] Thompson, P. M., "Digital Arithmetic Units for a High Data Rate," The Radio and Electronic Engineer, Vol. 45, No. 3, 1975.

[21] Gardner, P. L., "Functional Memory and Its Microprogrammed Implications," IEEE Trans. Comput., Vol. C-20, No. 7, July 1971, pp. 764-775.

[22] Lee, C. Y. and Paull, M. C., "A Content Addressable Distributed Logic Memory with Application to Information Retrieval," Proc. IEEE, Vol. 51, June 1963, pp. 924-932.

[23] Crane, B. A. and Githens, J. A., "Bulk Processing in Distributed Logic Memory," IEEE Trans. EC, April 1966, pp. 186-196.

[24] Batcher, K. E., "STARAN Parallel Processor System Hardware," National Computer Conference, AFIPS Proc., 1974, pp. 405-410.

List of References (continued)

[25] deRegt, M. P., "Introduction to Negative Radix Number Systems," Part I, Computer Design, May 1967, pp. 53-63.

[26] Avizienis, A., "Signed-Digit Number Representation for Fast Parallel Arithmetic," IRE Trans. EC, Vol. EC-10, Sept. 1961, pp. 389-400.

[27] Shaipov, N. Yu., "Methods of Realizing Arithmetic Operations in the Minus-Two Number System," Automation and Remote Control, 1970, pp. 835-841.

[28] Avizienis, A. and Tung, C., "A Universal Arithmetic Building Element (ABE) and Design Methods for Arithmetic Processors," IEEE Trans. Comput., Vol. C-19, No. 8, Aug. 1970, pp. 733-745.

[29] Avizienis, A. and Tung, C., "Design of Combinational Arithmetic Nets," Digest 1st Annual IEEE Computer Conference (Chicago, Illinois), Sept. 6-8, 1967, pp. 25-28.

[30] Pisterzi, M. J., "A Limited Connection Arithmetic Unit," Ph.D. dissertation, Department of Electrical Engineering, University of Illinois, Urbana, Illinois; also, DCS Report No. 398, June 1970.

[31] deRegt, M. P., "Negative Radix Arithmetic, Part 4, Multiplication and Division," Computer Design, Aug. 1967, pp. 36-44.

[32] _____, "Negative Radix Arithmetic, Part 5, Division: Testing the Remainder," Computer Design, Sept. 1967, pp. 44-50.

[33] _____, "Negative Radix Arithmetic, Part 6, Manual Division: the Magnitude Test," Computer Design, Oct. 1967, pp. 68-77.

[34] Szabo, N. S. and Tanaka, R. I., Residue Arithmetic and Its Applications to Computer Technology, New York, McGraw-Hill, 1967.

[35] Atkins, D. E., "Design of Arithmetic Units of ILLIAC III: Use of Redundancy and Higher Radix Methods," IEEE Trans. Comput., Vol. c-19, No. 8, Aug. 1970, pp. 720-733.

[36] Cristelly, R. de ORY, "Design of a Dynamically checked, Signed-Digit Arithmetic Unit," Computer Science Department, University of California, Los Angeles, California, Report No. UCLA-ENG-7366, November 1973.

[37] Sweeney, T., "An Analysis of Floating-Point Addition," IBM Systems Journal, Vol. 4, No. 1, pp. 31-42, 1965.

List of References (continued)

[38] Borovec, R. T., "The Logical Design of a Class of Limited Carry-Borrow Propagation Adders," M.S. Thesis, Department of Electrical Engineering, University of Illinois, Urbana, Illinois, August, 1968. Also, Report No. 275, Department of Computer Science, University of Illinois, Urbana, Illinois.

[39] Rohatsch, F. A., "A Study of Transformations Applicable to the Development of Limited Carry-Borrow Propagation Adders," Ph.D. Thesis, Department of Electrical Engineering, University of Illinois, Urbana, Illinois, June, 1967. Also, DCS Report No. 226, Department of Computer Science, University of Illinois, Urbana, Illinois.

[40] Robertson, J. E., "A Deterministic Procedure for the Design of Carry-save Adders and Borrow-save Subtracters," Department of Computer Science, University of Illinois, Urbana, Illinois, Report No. 235, July 5, 1967.

[41] Foster, C. C. and Stockton, F. D., "Counting Responders in an Associative Memory," IEEE Trans. Comput., Vol. C-20, pp. 1580-1583, December 1971.

[42] Bell, C. G. and Newell, Allen, Computer Structures: Readings and Examples, New York, McGraw-Hill Inc., 1971, pp. 628-637.

[43] Preparata, Franco P., "On the Representation of Integers in Non-adjacent Form," SIAM J. Appl. Math, Vol. 21, No. 4, December 1971.

[44] Avizienis, A., "Arithmetic Microsystems for the Synthesis of Function Generators," Proc. IEEE, Vol. 54, No. 12, December 1966.

[45] Goyal, L. N., "ILLIAC III Computer System Manual: Arithmetic Units-Vol. 2," Department of Computer Science, University of Illinois, Urbana, Illinois, Report No. UIUCDCS-R-73-551, January 1973.

[46] Texas Instruments Incorporated, TTL Integrated Circuits Catalog, Dallas, Texas Instruments Catalog CC201, August 1969.

[47] Kuck, D., Budnick, P., Chen, S. C., Davis, E., Han, J., Kraska, P., Lawrie, D., Muraoka, Y., Strebendt, R. and Towle, R., "Measurement of Parallelism in Ordinary FORTRAN Programs," IEEE Computer, January 1974, pp. 37-46.

[48] Knuth, D. E., "An Empirical Study of FORTRAN Programs," Software Practice and Experience, Vol. 1, pp. 105-133, April-June 1971.

List of References (continued)

[49] Foster, C. C. and Riseman, E. M., "Percolation of Code to Enhance Parallel Dispatching and Execution," IEEE Trans. Comput., Vol. C-21, No. 12, pp. 1411-1415, December 1972.

[50] Atkins, D. E., "A Study of Methods for Selection of Quotient Digits During Digital Division," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1970. Also, DCS Report No. 397, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1970.

APPENDIX A-1

ALGEBRAIC DESIGN OF A RIGHT-DIRECTED RECODER TO CHANGE MULTIPLIER DIGIT'S REDUNDANCY FROM $\delta = 1$ to $\delta \leq 2/3$[†]

This recoder changes the multiplier operand

$$M = 0 \cdot m_1 \, m_2 \, m_3 \cdots m_{j-1} \, m_j \, m_{j+1} \cdots m_n$$

where

$$|m_j| \leq (r-1) \qquad \forall j = 0,1,\ldots n.$$

to an algebraically equivalent operand

$$M' = m'_0 \cdot m'_1 \, m'_2 \, m'_3 \cdots m'_{j-1} \, m'_j \, m'_{j+1} \cdots m'_n$$

such that

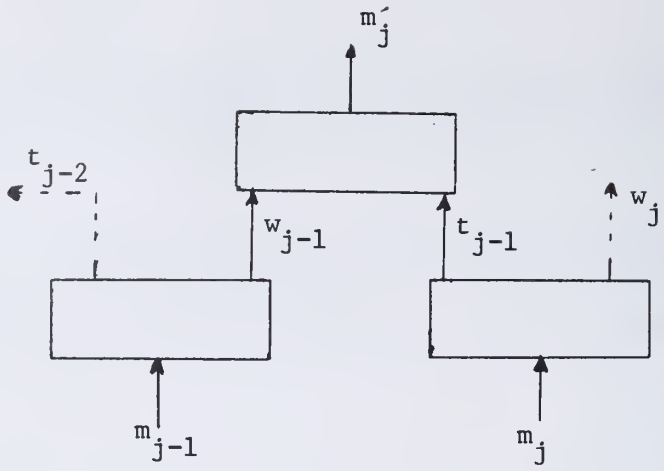
$$|m_0| \leq 1 \quad \text{and}$$

$$|m_j| \leq \left\lceil \frac{2}{3} (r-1) \right\rceil \quad \forall j = 0,1,\ldots n.$$

In order to do the above recoding serially on a digit-by-digit basis, starting from the most significant digit, one needs to know only the digit to the immediate left of the digit to be recoded in addition to the digit itself. For example, if m_j is the digit to be recoded, then the algebraic design of the recoder is given by the following

---

[†]Strictly speaking, $\delta = \left\lceil \frac{2}{3} (r-1) \right\rceil / (r-1)$ which may be slightly larger than 2/3.



Each digit m_j and m_{j-1} is first recoded into a pair of digits $\{t_{j-1}, w_j\}$ and $\{t_{j-2}, w_{j-1}\}$ so that

$$\begin{aligned} m_{j-1} &= r\, t_{j-2} + w_{j-1} \\ m_j &= r\, t_{j-1} + w_j \end{aligned} \quad , \quad r \geq 4$$

and

$$t_{i-1} = \begin{cases} 1 & \text{if } m_i \geq \left\lceil \frac{2}{3}(r-1) \right\rceil \\ 0 & \text{otherwise} \\ \bar{1} & \text{if } m_i \leq -\left\lceil \frac{2}{3}(r-1) \right\rceil \end{cases} \quad , \; i = j,\, j-1$$

$$-\left(\left\lceil \frac{2}{3}(r-1) \right\rceil - 1 \right) \leq w_i \leq \left\lceil \frac{2}{3}(r-1) \right\rceil - 1$$

The recoded digit m'_j is given by

$$m'_j = w_{j-1} + t_{j-1}$$

The above recoder is applicable for all values of index j. Note that m'_0 cannot have a magnitude greater than 1 and the recoded multiplier operand may have one digit extra compared to the original operand.

APPENDIX A-2

PRECISION REQUIREMENTS FOR QUOTIENT DIGIT CALCULATION

According to the analysis by Atkins [50] based on P-D plot considerations, the worst case precision of the operands required for quotient digit calculation is given by the relation

$$|\Delta P| \leq \hat{D}_t^{min} (\delta - \tfrac{1}{2}) - \frac{|\Delta d|}{2} (n-1+\delta) \qquad \text{(A2.1)}$$

where

ΔP = truncation error in the left shifted (by one digital position) partial remainder

Δd = truncation error in the divisor

n = maximum allowed value of the quotient digit

δ = redundancy ratio of the quotient digit

$= \frac{n}{r-1}$

and

$\hat{D}_t^{min}$ = minimum value of the truncated divisor

Let $|\Delta d| = r^{-\Omega}$

and $|\Delta P| = r^{-(\Omega-1)}$

where Ω = number of digits in the truncated divisor

Case 1: $\delta = 1$

For a maximally redundant quotient digit,

$\delta = 1$

$n = (r-1)$

and for a maximally redundant divisor normalized according to Definition 3 given in Section 3.5,

$$\hat{D}_t^{min} = \frac{r-1}{r^2} + \frac{1}{r^{\Omega}}$$

Substituting the above values in Equation (A2.1), we have

$$\bar{+}r^{(\Omega-1)} \leq (\frac{r-1}{r^2} + \frac{1}{r^{\Omega}})(1 - \tfrac{1}{2}) - \frac{r^{-\Omega}}{2}(r-1-1+1)$$

$$\leq (\frac{r-1}{r^2} + \frac{1}{r^{\Omega}})\tfrac{1}{2} - \frac{r^{-\Omega}}{2}(r-1)$$

which simplifies to

$$\bar{+}r^{\Omega} \leq \frac{(r-1)}{r^2(3r-2)} \qquad \text{(A2.2)}$$

Values of Ω which satisfies the relation (A2.2) for different values of r are tabulated in Table A.1.

Table A.1

Values of Ω Vs Radix and Redundancy Ratio of a Quotient Digit

Radix r	Redundancy ratio, δ, of a quotient digit	
	$\delta = 1$	$\delta \leq 2/3$
2	4	-
4	3	4
8	3	3
16	3	3
32	3	3
64	3	3

Case 2: $\delta = 2/3$

In this case,

$$n = \left\lceil \frac{2}{3}(r-1) \right\rceil$$

$$\hat{D}_t^{min} = \frac{r-1}{r^2} + \frac{1}{r^{\Omega}}$$

$$-r^{-(\Omega-1)} \leq \left(\frac{r-1}{r^2} + \frac{1}{r^{\Omega}}\right)\left(\frac{n}{r-1} - \tfrac{1}{2}\right) - \frac{r^{-\Omega}}{2}\left(n-1 + \frac{n}{r-1}\right)$$

which simplifies to

$$-r^{-\Omega} \leq \frac{2n-(r-1)}{r^2} \cdot \frac{r-1}{2r(r-1) + n(r-2)} \qquad \text{(A2.3)}$$

Values of Ω which satisfies the relation (A2.3) for different values of r are given in Table A.1.

Table A.1 clearly shows that 3 digits of the divisor and 2 most significant digits of the fractional part of the shifted partial remainder in addition to its (shifted partial remainder's) integer part are sufficient to calculate the quotient digit. It can be further shown that all the bits of the last digits of the truncated divisor and partial remainder are not necessary for the quotient digit calculation.

VITA

Lakshmi N. Goyal was born in Rohtak, India on October 19, 1941. He received the B.Tel.E degree in Electronics and Telecommunication Engineering in 1964 and M.Tel.E degree in 1965, both from Jadavpur University, Calcutta, India, and Ph.D. degree in Electrical Engineering from the University of Illinois, Urbana in 1976.

From 1963 to 1965 he was associated with Indian Statistical Institute - Jadavpur University joint computer project. He was responsible for the logic design and hardware implementation of the Arithmetic and Control Units of the Computer ISIJU-1, a variable word-length, microprogrammed solid state digital computer. In 1965, he became a lecturer in the Department of Electronics and Telecommunication Engineering, Jadavpur University, Calcutta, India, and continued to be asociated with the ISIJU-1 project. Since 1967, he has been a Research Assistant in the Department of Computer Science, University of Illinois, Urbana. From 1967 to 1971, he was associated with the Image Processing ILLIAC III Computer Project and worked on the design and implementation of the Scan-Display system, Interrupt Unit and Arithmetic Units of ILLIAC III. His research interests include Computer Arithmetic, Computer Architecture, Microprogramming, Digital System Design and Display Processors. He has several publications in these areas.

Mr. Goyal is a member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-76-797	2.	3. Recipient's Accession No.

Title and Subtitle
A STUDY IN THE DESIGN OF AN ARITHMETIC ELEMENT FOR SERIAL PROCESSING IN A LINEAR ITERATIVE STRUCTURE

5. Report Date: May, 1976

6.

Author(s)
Lakshmi Narayana Goyal

Performing Organization Name and Address
Department of Computer Science
University of Illinois
Urbana, Illinois

8. Performing Organization Rept. No.

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NSF DCR 73-07998

Sponsoring Organization Name and Address
National Science Foundation
Washington, DC

13. Type of Report & Period Covered

14.

Supplementary Notes

Abstracts This study is concerned with the design of an Arithmetic Element for Serial ocessing in a Linear Iteratively Structured Arithmetic Unit. The Arithmetic Unit is e up of identical logic modules called Processing Elements (PEs) such that each module ically communicates with a maximum of three of its neighboring modules for data and trol information. An arithmetic instruction is executed by a sequence of elementary roinstructions such that each microinstruction is executed by all the modules not in chro-parallelism, but in sequence by each module. The arithmetic processing takes ce serially on a digit-by-digit basis with the most-significant-digit-first (MSDF).

The arithmetic and logic design of the Processing Element and the implications of design choices on the LSI implementation of a PE is described. The MSDF nature of thmetic execution necessitates the use of the redundant number system for processing. arithmetic design of the PE is discussed with respect to the number representation, definition of a normalized number and the algebraic design of the digit algorithms the microinstructions necessary to implement the four basic arithmetic operations. mulas are given for the gate and pin complexities of the various components of the PE a function of the type of 2-tuple logic vector encoding for a redundant binary digit, bit width of the PE and the amount of redundancy in the multiplier/quotient digit. is found that a sign-magnitude logic vector encoding and the multiplier/quotient digit's redundancy of 2/3 or less should be employed in the design of the Processing Element.

KEY WORDS:

Addition
Arithmetic Design
Arithmetic Element
Digit-by-Digit Algorithms
Digital Computer Arithmetic
Distributed Memory
Division
Iterative Structure
Large Scale Integration
Multiplication
Processing Element
Redundant Number System
Serial Processing
Subtraction

17 Identifiers/Open-Ended Terms

17 COSATI Field/Group

18 Availability Statement

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page) UNCLASSIFIED

21. No. of Pages

22. Price